Task Parallel Assembly Language for Uncompromising Parallelism

Mike Rainey
Carnegie Mellon University
Pittsburgh, PA, USA
me@mike-rainey.site

Ryan R. Newton
Facebook
New York, NY, USA
Newton@fb.com

Kyle Hale
Illinois Institute of Technology
Chicago, IL, USA
khale@cs.iit.edu

Nikos Hardavellas
Northwestern University
Chicago, IL, USA
nikos@northwestern.edu

Simone Campanoni
Northwestern University
Chicago, IL, USA
simonec@northwestern.edu

Peter Dinda
Northwestern University
Chicago, IL, USA
pdinda@northwestern.edu

Umut A. Acar
Carnegie Mellon University
Pittsburgh, PA, USA
umut@cs.cmu.edu

Abstract

Achieving parallel performance and scalability involves making compromises between parallel and sequential computation. If not contained, the overheads of parallelism can easily outweigh its benefits, sometimes by orders of magnitude. Today, we expect programmers to implement this compromise by optimizing their code manually. This process is labor intensive, requires deep expertise, and reduces code quality. Recent work on *heartbeat scheduling* shows a promising approach that manifests the potentially vast amounts of available, latent parallelism, at a regular rate, based on even beats in time. The idea is to amortize the overheads of parallelism over the useful work performed between the beats. Heartbeat scheduling is promising in theory, but the reality is complicated: it has no known practical implementation.

In this paper, we propose a practical approach to heartbeat scheduling that involves equipping the assembly language with a small set of primitives. These primitives leverage existing kernel and hardware support for interrupts to allow parallelism to remain latent, until a heartbeat, when it can be manifested with low cost. Our Task Parallel Assembly Language (TPAL) is a compact, RISC-like assembly language. We specify TPAL through an abstract machine and implement the abstract machine as compiler transformations for C/C++ code and a specialized run-time system. We present an evaluation on both the Linux and the Nautilus kernels, considering a range of heartbeat interrupt mechanisms. The evaluation shows that TPAL can dramatically reduce the overheads of parallelism without compromising scalability.

CCS Concepts: • **Software and its engineering → Parallel programming languages**.

Keywords: parallel programming languages, granularity control

ACM Reference Format:
Mike Rainey, Ryan R. Newton, Kyle Hale, Nikos Hardavellas, Simone Campanoni, Peter Dinda, and Umut A. Acar. 2021. Task Parallel Assembly Language for Uncompromising Parallelism. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 31 pages. https://doi.org/10.1145/3453483.3460969

1 Introduction

A classic problem in parallel computing is to take a high-level parallel program, written in nested-parallel style, with fork-join constructs, and derive from it an executable that is efficient on real machines. Traditionally, solutions involve optimizing the program to control the amount of parallelism exposed, thereby limiting the overheads of task creation and scheduling [3, 7, 30, 60]. Left unchecked, task overheads can reach two orders of magnitude or more, effectively wiping out the benefits of parallelism. But when task overheads are addressed, the associated optimizations involve changing the code so that the program switches from parallel to sequential code, typically at "small" problem sizes. This is a process called *granularity control* [3, 7]. In addition to labor-intensive changes, granularity control requires tuning the program so that it switches from parallel to serial at appropriate points at run time. Such tuning may cause the program to lose its performance portability, because the process of tuning usually overfits to the idiosyncrasies of a particular machine. In particular, the notion of "small" depends on the machine architecture and software environment and varies significantly from one machine to another [60].

Motivated by the limitations of manual code optimizations and granularity control, recent work proposed an alternative approach that can, in principle, be completely automated. This approach, called heartbeat scheduling [5], provably controls the overheads of parallelism without requiring manual changes to the code. In heartbeat scheduling, a regular heartbeat event interrupts the program periodically to promote latent opportunities for parallelism into actual tasks that can be executed in parallel, e.g., by migrating across cores. Therefore, rather than making granularity decisions in the source code of the program, the program exposes the *maximum* amount of parallelism, and the heartbeat mechanism decides how and when to promote latent parallelism into actual parallelism. Intuitively speaking, because the heartbeat only fires at certain points in time, the cost of creating and managing parallelism can be amortized over the useful work done between heartbeats.

Although the idea of the heartbeat scheduling may appear quite simple, its realization is far from it. Perhaps the most important challenge is determining, at each heartbeat, a unit of work that must be reified as a task, so that it can be executed in parallel, just like any other task that is created by carefully hand-optimized codes. Prior work evaluates heartbeat scheduling by using a C++ interpreter that runs programs hand-rolled in a custom format, representing programs as abstract syntax trees. This structure allows the interpreter to create tasks by manipulating the abstract syntax tree at a heartbeat event. But runtime interpretive overhead is impractical, especially for high-performance parallel programming, and the interpreter provides only approximate timing of heartbeats.

In this paper, we propose an execution model for heartbeat scheduling, formalized by our Task Parallel Assembly Language (TPAL), and a runtime system that is backed by a practical heartbeat mechanism, which is implemented on top of hardware-based interrupts. The core of TPAL resembles a conventional, RISC assembly language. But unconventionally, TPAL features native support for task parallelism, consisting of a small collection of primitives and annotations on basic blocks. Because its task parallelism is native, TPAL can express parallel loops naturally and execute them efficiently. TPAL can achieve task parallelism as a *nearly zero-cost abstraction*, even with programs involving irregular, and nested parallel loops.

Because it is specified as an abstract machine, TPAL's execution model is low-level and detailed enough to be thought of as a model for an implementation. To evaluate this execution model, we present an implementation of TPAL along with a runtime system supporting the operations needed for implementing heartbeat events and parallel evaluation. The runtime system is written in C++, and is optimized to achieve practical efficiency by using state-of-the-art scheduling techniques. To implement the heartbeat events, the runtime system relies on hardware interrupts, which can be controlled at greater precision and lower cost than by using software-based approximations. Because the heartbeat events are hardware-driven, the runtime system is reasonably decoupled from the specific implementations of heartbeat events, and is quite portable, allowing it to be used on different hardware and software platforms.

To evaluate the effectiveness of proposed techniques based on TPAL coupled with hardware interrupts, we consider a Linux-based system and an experimental kernel framework, called Nautilus [34]. Nautilus is specifically designed to support parallel runtime systems, and thus allows tight control over hardware resources and elides many of the overheads and abstraction layers that arise from supporting general-purpose workloads.

For our evaluation, we consider a number of parallel benchmarks, including those that exhibit high degrees of irregular parallelism. Our results indicate that TPAL, driven by hardware-based interrupts, achieves excellent work-efficiency, incurring small overheads for uniprocessor executions, without sacrificing parallelism, and scaling well as the number of cores increases. Compared to Cilk, a state-of-the-art task-parallel system, TPAL is comparable or better, even as it manages all parallelism *automatically*, driven by hardware interrupts. Our results from experimenting with Nautilus show that there is room for improvement in the efficiency of Linux-based, software signaling mechanisms.

This paper makes the following contributions:

- TPAL: a Task-Parallel Assembly Language for task-parallel programming;
- An execution model for TPAL as an abstract machine that controls creation of parallel tasks automatically using hardware driven interrupts;
- A portable runtime system for TPAL, written in C++, that can be used to run TPAL programs on modern multicore hardware;
- An evaluation both on Linux and Nautilus, an experimental kernel framework for parallel runtimes.

2 Task Parallel Assembly Language

For heartbeat scheduling, we need the ability to interrupt a running program and examine its runtime state efficiently. To see how, consider the following simple loop, written in C, as a running example. It calculates the product of numbers passed in a and b (using addition), and leaves the result in c.

```
int a,b,c; int r = 0;
for (; a != 0; a--) { r += b; }
c = r;
```

To parallelize this loop with minimal cost, we ideally do not want to change *anything* about how the sequential version is compiled and executed. To this end, we propose to extend the assembly language with instructions that will allow executing the sequential code without any overheads, while, at the same time, making it possible to manifest latent parallelism. For example, the loop's index variable a is likely kept in a register rather than on the stack, and if we wish to take a heartbeat at runtime and *split* the remaining iterations (for parallelism), then we must find the current iteration in the appropriate register, rather than relying on finding that index in memory. This constraint bears resemblance to the one faced by a debugger, that is, in terms of interpreting runtime program state. But we cannot tolerate anything nearly as expensive as a debugger implementation (e.g., Linux ptrace).

Figure 2 shows the code for our running example in TPAL. In this code, if we assign the empty block annotations, $\star_{exit} = \star_{loop} = \cdot$, the first three blocks of the resulting code are identical to ones produced by a sequential C compiler. Apart from using jump statements, this code is similar to the C source code. Once this code executes, it is irrevocably sequential. But the unique ability of TPAL, among assembly languages [1], is that we can add parallelism *without* changing the sequential behavior, and in fact without changing the code, except by adding annotations. We mark *promotable* program points, where it is possible to switch from the sequential loop to a parallel variant, and then join again afterwards. More precisely, we derive the parallel version by assigning $\star_{exit}$ = **jtppt assoc-comm**; {r ↦ r2}; comb and $\star_{loop}$ = **prppt** loop-try-promote.

TPAL captures exactly the essential property of interruptability, and provides the compiler with building blocks to construct semantically equivalent parallel and sequential variants of a function, with the ability to redirect between them at runtime. Because TPAL is a general-purpose, abstract assembly language, it can be targeted by a wide range of parallel source languages and can in turn be lowered to existing ISAs or low-level intermediate representations. (We target x86-64 in our evaluation, Section 4.)

2.1 Syntax and Execution Model

The syntax of TPAL is presented in Figure 1. It is based on a subset of the MIPS assembly language, using a familiar

[1]The ability to elide parallelism annotations and have a semantically equivalent program is a goal shared by spawn/sync annotations in Cilk, or parallelism combinators in Haskell, but at the assembly level it is quite a different thing.

$r \in$ registers, $l \in$ labels,

$n \in$ integer literals, $j \in$ join-record identifiers

$$\begin{array}{lcl}
v & ::= & r \mid l \mid n \mid j \\
op & ::= & + \mid - \mid \ldots \\
\iota & ::= & r := v \mid r := op\ r, v \mid \textbf{if-jump}\ r, v \\
 & \mid & r := \textbf{jralloc}\ v \mid \textbf{fork}\ r, v \\
I & ::= & \textbf{jump}\ v \mid \iota; I \mid \textbf{halt} \mid \textbf{join}\ v \\
B & ::= & [\,\star\,]\ I \\
\star & ::= & \cdot \mid \textbf{prppt}\ l \mid \textbf{jtppt}\ jp; \Delta R; l \\
jp & ::= & \textbf{assoc} \mid \textbf{assoc-comm} \\
\Delta R & ::= & \{ r_1 \mapsto r_1', \ldots, r_n \mapsto r_n' \}
\end{array}$$

Figure 1. Grammar of TPAL. Highlighted syntax is specific to our parallel extensions, whereas the rest represents a conventional RISC instruction set.

```
prod: [·] // computes c =  a * b
  r := 0; jump loop
exit: [★exit]
  c := r; halt
loop: [★loop]
  if-jump a, exit;
  r := r + b; a := a - 1;
  jump loop
loop-try-promote: [·]
  t := a < 2; if-jump t, loop;
  jr := jralloc exit;
  jump loop-promote
loop-par-try-promote: [·]
  t := a < 2; if-jump t, loop-par;
  jump loop-promote
loop-promote: [·]
  m := a / 2; n := a % 2; a := m;
  tr := r; r := 0;
  fork jr, loop-par;
  a := m + n; r := tr;
  jump loop-par
loop-par: [prppt loop-par-try-promote]
  if-jump a, exit-par;
  r := r + b; a := a - 1;
  jump loop-par
comb : [·]
  r := r + r2; join jr
exit-par: [·]
  join jr
```

Figure 2. The prod program in TPAL. Applying the empty block annotations, $\star_{exit} = \star_{loop} = \cdot$, yields a serial program. Applying $\star_{exit}$ = **jtppt assoc-comm**; {r ↦ r2}; comb and $\star_{loop}$ = **prppt** loop-try-promote, yields a parallel program.

notation for instructions for readability. In TPAL, the execution of a program consists of a set of concurrent tasks, such that each task has its own private register file and call stack. Heap memory can be shared. How the tasks themselves are scheduled, that is, the order in which tasks run (up to dependency constraints) and on which core they run, is up to the *load-balancing algorithm*. TPAL is agnostic to load-balancing algorithm: it is compatible with e.g., variants of work stealing [6, 16, 19] or parallel depth first [46].

We start with the subset of the language that supports register-based memory, and later address the stack and heap. Our assembly language assumes a set of registers, r, labels l, integer literals n, and a special set of values that we call *join records*. A join record j is memory that is used by TPAL programs to synchronize multiple tasks at a join point. An operand v is either a register, a label, an integer literal, or a join record. A primitive operation op is one of a number of primitive operations that can be found on a conventional RISC machine, such as arithmetic operations for integers.

An instruction $\imath$ is either a move-to-register operation, a primitive operation, a conditional jump, a join-record allocation, or a fork instruction. A join-record allocation instruction allocates (on the heap) and initializes a new join record. Its argument is a label that is to be the *continuation block* of the join point. A fork instruction spawns a new task, in a fashion resembling that of the UNIX `fork()` system call. It takes two arguments: first, a join record, and, second, a label from which the spawned task is to start executing. We call the spawned task the *child task* and the calling task the *parent task*. When it executes, the first action of the fork instruction is to register the dependency edge between the parent and child task in the join record. After the dependency is registered, the child task is added to the set of executing tasks, from which point it starts executing with a copy of the register file of its parent. The child task starts by executing the block at the label passed for the second argument of the fork instruction. After it issues the fork instruction, the parent task proceeds to issue its next instruction.

An instruction sequence I is a list-based representation of a sequence of assembly instructions: it is either a jump instruction, a sequencing operator (semicolon), a halt instruction, or a join instruction. A jump instruction is an unconditional jump operation. A semicolon operator specifies a sequential order between an instruction and an instruction sequence. A halt instruction terminates the whole machine. A join instruction initiates synchronization between a parent and one of its child tasks. Its first argument is a join record. When it executes this instruction, a task participates in a *join-resolution policy*. A join-resolution policy specifies the manner in which to combine the results held in the memories of the parent and child tasks. Upon completion of a join, that is, after all tasks registered in the join record issue their join instructions, the program then jumps to the label originally passed in the allocation of the join record.

A program is represented by a set of (labeled) code blocks. Each code block consists of an instruction sequence, along with an annotation. Such annotations, denoted by $\star$, are either an empty annotation, a promotion-ready program point, or a join-target program point. A *promotion-ready program* point is the entry point of a block for which there is a special behavior: when control targets the block, either control can flow, as usual, into the first instruction of the block, or control can flow, instead, to the label attached to the annotation. A *join-target program point* is the entry point of a code block that is assigned to be the continuation of the join point of parallel tasks. The join-target annotation specifies the join-resolution policy, i.e., the instruction sequence to be executed upon parent and child tasks meeting at their common join points. Its first component jp specifies whether the combining operation is only associative or both associative and commutative. Its second component ΔR specifies the way in which joining tasks combine their register files into one register file, which is to be used by the combining block. Its third component specifies the label of the combining block.

2.2 Dynamics

We designed TPAL to make it natural and efficient to implement heartbeat scheduling: parallelism is introduced on a regular basis, in a two-stage process. The first stage involves the triggering of an interrupt and the second involves the manifestation of latent parallelism by the interrupt handler, which may fork a new task. The triggering of an interrupt is supported in TPAL by assigning each task a cycle counter, which increments every time a task issues an instruction. When the cycle counter of a task exceeds a certain threshold, that task is ready to trigger an interrupt. The threshold is a global parameter, written $\heartsuit$, and is determined by a one-time, per-machine tuning process, which is required by heartbeat scheduling [5]. The setting of the parameter is picked by the heartbeat tuner application to be just large enough to amortize the creation of a new task, but small enough to avoid pruning away useful amounts of parallelism.

Adding Parallelism. We return to our running example program, `prod`. By using the parallel block annotations, the program will ultimately allow the loop to be parallelized on demand. The serial-by-default structure in the program is its main strength: it is the reason we can achieve near zero-cost abstraction.

Heartbeat interrupts. When the heartbeat threshold is exceeded by a task, an interrupt is ready to be serviced the next time the program enters a promotion-ready program point. In our example, every time a task enters the `loop` block and its heartbeat threshold has passed, the task jumps to its handler block, namely `loop-try-promote`, instead of the first instruction of the `loop` block. The handler block checks on line 10 if there is any parallelism available in the remaining iterations of the loop. If there is no latent parallelism, then our running task jumps back to the `loop` block, where it left off, but if there is, the task manifests the parallelism.

Promotion. To manifest the parallelism from the loop, our task performs a *promotion*, wherein a task creates a join point and forks a new, child task. Promotion begins in our running example on line 11, where our task allocates a join point. By passing to the join-allocation instruction the `exit` label, we are instructing our newly parallelized loop to terminate, with the final result, by jumping to the `exit` block. The handler next jumps to the `loop-promote` block, where our task creates parallelism from the loop induction variable `a`. The parallelism is created by dividing up the remaining iterations into two parts, with half going to the child task, and half to its parent. From this point, both parent and child tasks proceed to work on their respective parts of the remaining iterations, but now start from a new block, namely `loop-par`.

Parallel tasks. Now that they are running in parallel, our parent and child tasks have to complete their own local computations, and then combine their local parts of the overall result. To perform their local computations, our parallel tasks execute the `loop-par` block, which performs the same steps as the `loop` block (except for line 23). While our parallel tasks execute, additional heartbeat interrupts may trigger additional promotions, thereby recursively manifesting latent parallelism from the tasks. All heartbeat interrupts from hereon jump to the `loop-par-try-promote` handler block, thereby ensuring that all future tasks share the join record in `jr`. After any one of our parallel tasks completes its local work, the task exits (on line 23) by branching to the `exit-par` block. In this block, the task enters the join-resolution protocol, where all parent and child tasks eventually meet with their partners to combine their local results (in the register `r`).

Join resolution. While the program is executing, the TPAL runtime keeps a record of the tree induced by the fork instructions issued by tasks. This bookkeeping tree is used by the join instruction to match each task with its parent or child. When a task issues the join instruction, it stashes its register file in the join record and removes the dependency edge on the join point it shares with its partner in the tree. The first task to complete this step terminates and removes itself from the set of running tasks, and the second performs the next step of join resolution. In this step, the runtime system seeds the task with a new register file. The new register file is obtained by taking the register file of the parent task and extending it with some entries from the register file of the child task. In our `prod` example, this merging process enables the child task to share with its parent the value in its accumulator register `r`. The annotation on the `exit` block specifies that the contents of register `r` from the child task be copied to register `r2` in the new register file.

After merging register files, the runtime schedules the parent task with this new register file, starting from the *combining block*, which is specified as `comb` in the annotation in our example program. Our combining block takes the sum of the accumulator variables from the parent and child tasks, puts the result in register `r`, and exits by issuing the join instruction again. This time, the join instruction either repeats the process one level up in the tree of tasks, or it reaches the root. If at the root, the join instruction jumps to the original target of the join record, which in our example is the `exit` block. At this point, the parallel program has completed its work.

2.3 Nested Parallelism

TPAL can express in a natural way various forms of nested parallelism, for example, in the form of nested loops and recursive functions or their combination. For example, we can implement a "power" function by nesting our running example, `prod`, inside an outer for loop. We parallelize the for loops by using the *outer-loop first* policy, a policy that requires that parallelism is created where it is most beneficial first, that is, from the least recent parallel context. This policy is a necessary condition for any implementation to be backed by the formal efficiency guarantees proved for heartbeat scheduling [5]. As we describe in our evaluation section, TPAL shines in its ability to efficiently execute nested loops, even when their workloads are irregular. The reason is that TPAL can always amortize the cost of parallelism, even across loop boundaries. We present full details of how such a nested parallel programs (including both loops and recursive functions) may be expressed in TPAL in the Appendix.

3 Implementation

This section describes the compiler and runtime/OS support needed to translate and execute high-level programs (e.g., Cilk Plus) down to assembly (e.g., x64).

3.1 Compiling from C++ to TPAL

To compile a high-level, parallelized loop down to the level of our TPAL assembly, we need to generate the sequential and parallel versions of a loop body, and represent them in the compiler so that the former can be promoted to the latter on-demand. We present our technique by returning to our running `prod` example, this time starting from a Cilk Plus version.

```
void prod_cilk(int a, int b, int* c) {
  reducer_opadd<int> r(0);
  cilk_for (; a != 0; a--) { r += b; }
  *c = r; }
```

The program uses the syntactic parallel-loop extension provided by Cilk Plus to compute the result. Its loop body uses Cilk's idiomatic pattern for accumulating results, namely reducer variables [29]. There are similar mechanisms in OpenMP [53] and TBB [41]. The reducer variable enables tasks to work independently on their own local view of r. When tasks join their results, they sum their local views to compute the final result.

In any such linguistic parallelism, there is necessarily some representation of high-level parallel constructs, e.g., cilk_for, reducer, that stays intact for some subset of the early stages of the compilation pipeline. Then, at some later stage, the high-level parallel constructs are lowered by that stage into simpler forms. However, the code resulting from lowering pass can easily block optimizations in subsequent passes, such as loop-invariant code motion, loop vectorization, etc. As such, there is motivation to keep the high-level structure and make optimization passes aware of it. This approach is exemplified by recent work on Tapir, an effort to extend LLVM's IR with support for Cilk-style parallelism [56]. Tapir carefully extends LLVM to allow high-level parallel constructs to propagate late into the compilation pipeline, thereby unlocking compiler optimizations that were otherwise inaccessible. Although our TPAL does not yet have compiler support, we believe that a new implementation combining Tapir and TPAL is feasible, and can benefit from the best of both worlds: (1) traditional compiler optimization of high-level parallel code for early stages, thanks to Tapir, and (2) efficient, granularity control, a la TPAL, for later stages. The idea is to implement a new pass that lowers high-level parallel constructs into TPAL instructions. We summarize the lowering steps below.

Code versioning. We show in Figure 4 the fragments needed for our prod example. The first piece of code is the prod function, which represents the initial serial-by-default part of the loop. It corresponds to the first three assembly blocks shown in Figure 2.

The parallel fragment needs to make use of the TPAL runtime system. We show its interface in Figure 3. The interface exports definitions for join-record objects, and fork and join functions, corresponding to the linguistic forms our TPAL formalism.

The next code fragment, namely prod_par, corresponds to the parallel blocks of assembly. It represents one chunk of work, executing sequentially on one processor, but in the context of a parallel execution. After finishing its local work, the function enters the join protocol by calling the join function. The final fragments of prod are its heartbeat handlers. For convenience, we changed the conventions relating to the fork instruction slightly compared to its counterpart in the formalism. Our fork instruction takes, in addition to the join record, closures corresponding to the child and parent tasks, and the combine block. Implicit in this convention is that the parent task is rematerialized by the handler function (and correspondingly, the interrupted task exits early to avoid duplicating work).

```
class joinrec { ... };
template <class Child, class Parent, class Comb>
void fork(joinrec* jr, Child c, Parent p,Comb m);
void join(joinrec* jr);
```

Figure 3. Scheduling interface used by application code scheduled by TPAL runtime.

```
void prod(int a, int b, int* c) {
  int r = 0;
  for (; a != 0; a--) { r += b; }
  *c = r; }
void prod_par(int a, int b, int* c, joinrec* jr) {
  int r = *c;
  for (; a != 0; a--) { r += b; }
  *c = r;
  join(jr); }
bool loop_try_promote(int a, int b, int* c) {
  if (a < 2)
    return false;
  joinrec* jr = new joinrec;
  loop_promote(a, b, c, jr);
  return true; }
void loop_promote(int a, int b, int* c,
                      joinrec* jr) {
  int m = a / 2; int n = a % 2;
  int* rs = new int[2];
  fork(jr,
    [=] { // child
      rs[1] = 0; prod_par(m, b, &rs[1], jr);
    }, [=] { // parent
      rs[0] = *c; prod_par(m + n, b, &rs[0], jr);
    }, [=] { // combine
      *c = rs[0] + rs[1];
      delete [] rs; join(jr)
    }); }
```

Figure 4. Code fragments of our prod program.

3.2 Safe & Efficient Heartbeat Triggering

Next, we describe the compiler, runtime, and OS support to enable the serial version of a loop body to be promoted to our parallel version for the next chunk of iterations.

Rollforward compilation In our formal model of TPAL, we assume that heartbeat handlers are triggered every time a task meets two conditions: (1) at least ♡ cycles passed since the previous handler invocation and (2) the control flow enters a promotion-ready program point. Although we cannot rely on any ready-made mechanism from the OS, we can build one on top of OS signaling. There is a challenge, however: given that an interrupt triggered by an OS signal may arrive at *any* step of execution of a task, we somehow need to satisfy the second condition above. In other words, we need a mechanism that triggers a heartbeat interrupt downstream from wherever the interrupted task might currently be executing, given just the current program context. Fortunately, there is a ready-made solution for this problem: we can use the classic technique of *rollforward compilation* [49]. Rollforward compilation is a general technique for protecting sections of code from being interrupted by OS signals. The idea is to compile a sequence of instructions so that, when preempted by a signal, the sequence executes to its end, and then invokes the signal handler. Our insight is that we can employ rollforward for parallel loops by treating as critical sections the control paths that exist between promotion-ready program points.

We implemented a small rollforward compiler for x64 assembly. The output of this compiler consists of two versions of the input program: original and the rollforward versions. The original version is a copy of the input assembly, modified only so that each line is labeled, e.g., `o0`, `o1`, and so on. As such, there is negligible runtime overhead cost the original program. The rollforward version differs from the original in two respects. First, it has different line labels, e.g., `r0`, `r1`, and so on. Second, any instruction in the rollforward version that jumps to a promotion-ready program point jumps instead to the corresponding handler function. The overall effect is that the original and rollforward instructions align perfectly up to instruction labels, the original version behaves such that it never triggers a heartbeat interrupt, and the rollforward version such that it always triggers a heartbeat interrupt at the next promotion-ready program point.

Enabling promotions To enable rollforward at runtime, we need to assign the TPAL worker threads a (Linux) signal handler. When it initializes, the TPAL runtime configures an alarm to invoke this interrupt handler every ♡ microseconds. When it is invoked, the handler uses a table that maps from labels in the source program to labels in its rollforward version. e.g., for x64 `prod` blocks, $\{\texttt{o0} \to \texttt{r0}, \ldots, \texttt{o7} \to \texttt{r7}\}$. This table is generated by the compiler, and is loaded once, by the binary load routine, before the TPAL runtime initializes. When it receives a signal, the handler inspects the program counter of the OS thread that was interrupted by the signal. If the program counter matches a key in the table, the handler replaces that program counter by the corresponding rollforward entry. As a consequence, the program will continue executing until it enters the next promotion-ready program point, at which time it will invoke a handler function.

In general, to make it safe to invoke a handler function, we need support from the compiler to generate *compensation code*. Compensation code is needed because compiler optimization passes may create drift between the program variables and the arguments expected by the handler function. The compensation code materializes the live variables from the registers and stack at the promotion-ready program point, and calls the handler function. For example, suppose we compile a program that iterates over an array, using a pair of integer indices to point to the next cell of the array and the length of the array, respectively. A compiler optimization may change the types of integer indices used in an array traversal to be direct pointers on the array, thereby creating an incompatibility with a handler function, which may expect as arguments the integer representation of the indices. This problem exists in many other contexts, such as debugging and JIT compilation, and fortunately, there is a general approach for solving it, namely *on-stack replacement (OSR)* [23]. Prior work shows that OSR does not significantly degrade code quality when there are a small number of points in the code that require replacing the stack [27, 39]. This is our case where we only have one promotion-ready program point per parallelized loop.

3.3 Taming Code Bloat

Our compilation technique causes an increase in the size of the program binary. Overall, the increase is in linear proportion to the size of the *parallel* regions of code in the program, i.e., blocks of code containing spawn/sync calls and parallel-for loops. This increase in code size can increase instruction-cache misses, but the misses remain under control, because transfers to the rollforward code are amortized by the heartbeat.

The potential blowup in code size due to the introduction of serial, parallel, and handler blocks can be controlled by compiler support. The handler blocks are likely to impose only marginal cost, whereas the issue of emitting different serial and parallel blocks is a bit more subtle, but nevertheless introduces only a modest tradeoff between the advantage of having different blocks for serial and parallel loop bodies, e.g., `loop` and `loop-par`, versus using one block that is the merging of `loop` and `loop-par`.

3.4 Benchmark Implementation

For our benchmark implementations, we used a manual version of on-stack replacement. We use existing compiler mechanisms to generate the compensation code required for each of our benchmark programs. In Figure 5, we demonstrate this technique with our `prod` program. The first step is to declare a flag, namely `heartbeat`, to stand in for an interrupt delivery. If the conditional branch at line 5 sees `true`, then the program invokes the handler function. However, we intervene so that this conditional branch is eliminated and therefore never executes at runtime (so we never need to allocate memory for the `heartbeat` global). We simply compile this C++ code to assembly, pass the assembly through our rollforward compiler, and complete the process by manually editing the assembly code generated by rollforward. The edits consist of eliminating from the final assembly all of the conditional branches for the `heartbeat` global. We replace each such conditional in the non-rollforward blocks with `nop` instructions, and we replace each such conditional in the rollforward blocks with an unconditional jump to the handler function. These changes together achieve the desired behavior: the non-rollforward part of the program skips over

```
extern volatile bool heartbeat;
void prod(int a, int b, int* c) {
  int r = 0;
  for (; a != 0; a--) { r += b;
    if (heartbeat &&
      (*c = r; loop_try_promote(a, b, c)))
      return; }
  *c = r; }
```

Figure 5. The instrumented version of our prod program, using our semi-manual rollfoward compilation technique.

the conditional, whereas the rollforward version jumps to a certain, compensation block. The compensation block materializes all the program state from the running program, e.g., prod, and calls the corresponding handler function, e.g., loop_try_promote. This instrumentation has close to zero cost in the common case, excluding any indirect costs associated with the nop instructions, which can be avoided with compiler support.

Signaling in Linux. In Linux, there are at least two off-the-shelf mechanisms we can leverage for heartbeat signals. The first one we call the *ping thread* because it employs a dedicated OS thread to send heartbeat signals to the worker threads. While the approach is simple, the linear signaling does not scale with large core counts, and unfortunately the pthreads library does not offer a broadcast/multicast interface. Some modern architectures expose programmable interrupts based on hardware performance counters. For example, the PAPI library [50] allows for interrupts to be raised on a per-core basis when the cycle counter on the appropriate core exceeds some programmer-specified threshold. Our implementation of TPAL can be configured to use the simple ping-thread approach or the Linux-based PAPI approach. Neither of these interfaces is a natural fit for our needs and neither is particularly optimized for low-latency delivery. Both software and hardware overheads in signal delivery have tangible effects on the achievable heartbeat frequency. In Section 4, we mitigate these effects using custom OS support.

4 Evaluation

We compared our TPAL-based Heartbeat Scheduling implementation with the state-of-the-art Cilk Plus system using a common set of benchmarks on a 16 core machine. This comparison is complex because, while Cilk's execution model performs an initial decomposition when latent parallelism is encountered, TPAL's execution model involves recurrent decomposition on each beat. It is also important to understand that, beyond the additional overheads incurred by TPAL, the amount of latent parallelism and whether it makes sense to manifest it, varies from benchmark to benchmark. Our goal is to effectively leverage the latent parallelism when it exists and is useful, while paying no cost when it does not exist or is not useful.

Our results show that TPAL creates tasks with considerably lower overhead than Cilk (geomean 13.8× lower), and hence can manage the finer granularity tasks that necessarily result from recurrent decomposition. At the maximum available scale, TPAL achieves significant speedups over Cilk (geomean 53%) for benchmarks that are amenable to recurrent decomposition, while the others (that do not lend well to our technique) incur minimal slowdown (geomean 9.8%).

Of course, if TPAL's overheads were zero, no slowdown would ever occur, and speedups could be enhanced. We next consider TPAL's overheads in detail. TPAL's compile-time transformations do not create significant overhead in the transformed code compared to original sequential code. The primary sources of overhead are due to the promotion process, and the heartbeat interrupt mechanism that triggers it. This section uses Linux signals as the mechanism. In the next section, we consider other mechanisms in pursuit of lower-overhead and finer-granularity heartbeat interrupts.

4.1 Benchmarks

Iterative (loop-based) benchmarks. We ported the *kmeans* and *srad* benchmarks from the Rodinia benchmark suite [21]. For *kmeans*, we use an input of 1 million objects, and for *srad* an 4k × 4k input matrix. The *spmv* benchmark is the classic sparse-matrix by dense-vector product algorithm. Its sparse-matrix input is represented in the compressed sparse row (CSR) format, with non-zero elements represented by double-precision floats. The random matrix is a sparse matrix with 273 million non-zero elements (and non-empty), and a maximum column size of 100. The powerlaw matrix is a random matrix with 186 million non-zero elements, with a power law characteristic [52]. Its largest column contains 5 million non-zero elements, which is 3% of the total number of non-zero elements in the matrix. The arrowhead matrix is a structure that is noted for being particularly challenging for task scheduling [59]: its diagonal, first column, and first row are filled with non-zero elements. The *floyd-warshall* benchmark is a purely loop-based implementation of the classic algorithm for finding the shortest path in a weighted graph. The *mandelbrot* benchmark computes a square image representation of the mandelbrot set [22].

Recursive benchmarks. We ported the *knapsack* and *mergesort* benchmarks from the Cilk benchmark suite [44]. The *knapsack* benchmark is the only one of our benchmarks that is non-deterministic: the amount of work it performs depends on the schedule. The *mergesort* algorithm is the only benchmark that uses both parallel loops and parallel recursive calls. In particular, the outer sort function and the inner merge function expose parallelism in a recursive, divide-and-conquer fashion, but there is also a parallel copy operation that moves items to and from a temporary buffer, which exposes parallelism via a parallel loop. The inputs of *mergesort* are generated from uniform and exponential distributions.

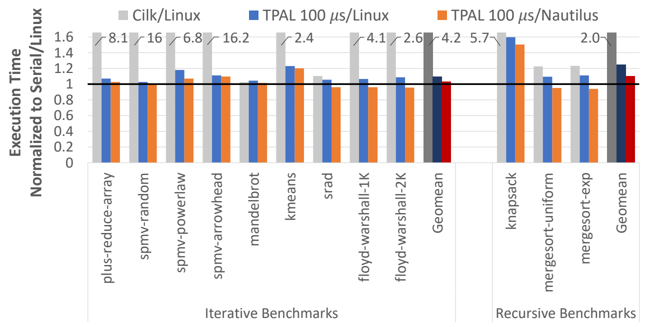


Figure 6. Task creation overheads for Cilk Plus and TPAL. In contrast to Cilk Plus, TPAL's overheads are minimal.

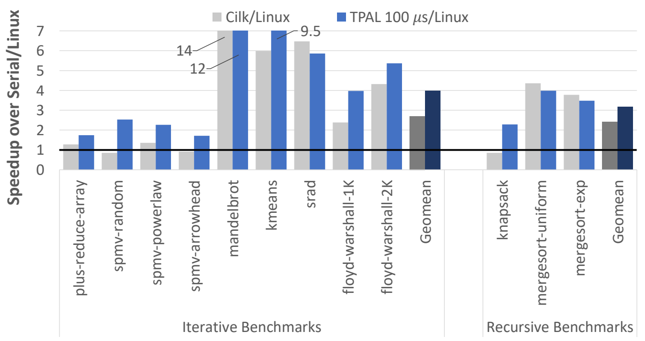


Figure 7. Speedups over serial execution on Linux, 15 cores. Overall TPAL outperforms Cilk Plus.

4.2 Experimental Setup

Our primary test bench is a Dell PowerEdge R6415, with a single-socket, 16-core AMD EPYC 7281 (Naples) processor running at 2.7GHz with 64KBL1i, 32KBL1d and 512KB L2 per core, and 32MB of shared L3 cache. It has one NUMA node and 32GB of DDR4 2400MHz RAM. Our machine runs Fedora Server 32, with stock Linux Kernel 5.8.13-200. We disabled SMT (hyperthreading) and we configure the machine's BIOS to use the maximum performance profile (thus disabling DVFS). For our test machine, we assigned the heartbeat rate to be ♡ = 100μs by following the tuning process proposed in the original heartbeat paper [5]. Our code is compiled with GCC version 7.5.0 with flags `-O3 -m64 -march=x86-64`. For load balancing, our TPAL runtime and that of Cilk Plus use randomized work stealing.

We reserve the first core either to do nothing during the benchmark or to execute the ping thread, when needed. The reasons we picked this configuration are (1) we wanted to avoid the overheads generated by a ping thread from affecting any of the worker threads that executed the benchmark workload and (2) we can avoid various other sources of overhead that can, in Linux and Nautilus kernels, slow down the first core in the system.

The Serial/Linux programs are, in all cases but one, the versions of the parallel programs with spawns/joins (also parallel loops) removed. Only in the case of mergesort did we use a significantly different serial program, i.e., serial mergesort. We report the average over 30 runs.

4.3 TPAL vs Cilk Plus

TPAL's task creation overheads are lower than Cilk's. Cilk Plus benefits from significant engineering to make its parallel function call mechanism efficient, even when the program runs on a single core [30]. Furthermore, its parallel loop construct implements an additional form of granularity control by splitting its parallel loop range into $8P$ blocks, where P denotes the number of cores. The single-core execution of Cilk Plus is nevertheless slowed down noticeably by task-creation costs in certain cases, whereas in our approach, there is one uniform mechanism, namely the heartbeat, that ensures task-related overheads are well amortized *for all programs and all inputs*. Figure 6 compares the single-core running times of Cilk Plus and TPAL versions of the benchmarks. In all cases but one, our implementations are as fast or faster than those of Cilk Plus. The only exception is *mandelbrot*, which is 2% slower.

TPAL performs better than Cilk at full scale. To scale up, both implementations need to create or promote a number of tasks that is sufficient to keep the cores fed, while also keeping task overheads low. Figure 7 compares the 15-core running times of Cilk Plus and TPAL versions of the benchmarks. The parallel execution times of the TPAL versions are considerably lower than those of Cilk, with four exceptions. Of these exceptions, only *mandelbrot* rises above 10%. When TPAL's recurrent decomposition comes into play, speedups of 53% (geomean) compared to Cilk result, while when recurrent decomposition does not come into play, slowdowns of only 9.8% (geomean) occur.

The *floyd_warshall* benchmark makes for an interesting point of comparison, because it shows a situation where Cilk's granularity-control heuristic for `cilk_for` loops fails. For the input size of 1k vertices, there is not enough parallelism to keep all 15 cores fed. The Cilk heuristic generates for this input a much larger number of tasks than our technique (23× more). As a consequence Cilk Plus achieves a higher utilization than TPAL (82% vs. 54%, respectively), and yet is 67% slower (Figure 7), owing to high task overheads. In effect, the Cilk version keeps processors fed doing the busy work of creating and destroying an overabundance of tasks, and ultimately performs worse for it. Our approach finds the right balance for this input, creating just enough tasks to keep cores fed with useful work, but not wasting time sharing too-small tasks. Moreover, as the input increases to 2K vertices, thereby favoring Cilk's granularity heuristic, our approach scales as well as Cilk does, taking advantage of an amount of parallelism that is closer to keeping all cores fed and reaches comparable utilization levels (89% for Cilk Plus vs. 84% for TPAL). We redirect the interested reader to the Appendix for more details.

Overall, these results show that our approach achieves excellent performance both on parallel as well as sequential runs, i.e., scaling up and down.

4.4 TPAL Approaches Near-Zero Cost

We now consider the sources of overhead that could limit the performance of TPAL. If these overheads were zero, we would expect the recurrent decomposition of TPAL to always perform strictly better than the initial decomposition of Cilk.

TPAL's compilation-related performance overhead is low. Thanks to its serial-as-the-default scheduling policy, our compilation technique produces binaries whose performance is close to that of corresponding serial programs, because they are very close to the serial programs. Figure 8 compares the execution time of the serial baseline programs of our benchmarks with those of our TPAL binaries. Here, the heartbeat interrupt mechanism is turned off, so pure sequential execution occurs. Our programs are at most 6% slower than their serial baseline programs, except for *floyd-warshall*, *kmeans*, and *knapsack*. The performance of *floyd-warshall* may be related to our manual compilation technique having to slightly modify the innermost loop which is very fine grained, causing a perturbation. In an integrated compiler-based implementation of TPAL (Section 7), that would be easily avoided. The *kmeans* benchmark is slower by 17% because the TPAL version uses an auxiliary data structure to accumulate centroid values, whereas the serial program does not. This situation is the same in the original Rodinia implementations, and as such is not a limitation of our compilation technique.

The 51% slowdown of *knapsack* is the most concerning. It happens for a simple reason: although this benchmark performs almost no computation besides recursive calls, our implementation still pays a cost for pushing and popping promotion-ready stack marks. This cost is visible in *knapsack*, because there is little other computation. In contrast, although it also incurs the costs of maintaining the promotion-ready stack marks, *mergesort* shows only 4–6% overhead, suggesting the bookkeeping costs are less significant. Additional optimizations (e.g., 32-bit pointers) may reduce this overhead but are outside the scope of this work.

Signal overhead is low, but could be improved. To promote latent parallelism in Heartbeat Scheduling, signals must interrupt each core on a regular basis. These interrupts need to arrive often enough to create sufficient parallelism, but far enough apart to keep overheads low. Figure 9 isolates and presents the overheads on a single core due to the interrupt mechanism only (i.e., without any promotions), as well as when interrupts generate parallel tasks. The bars labeled Serial represent runs of the *serial baseline* program (not TPAL), and therefore help to isolate the cost of interrupts. The figure presents results only for the INT-PingThread approach of producing the beat, as described in Section 3.2. We do not present the INT-Papi approach as it always incurs much higher overheads and does not provide any additional benefits.

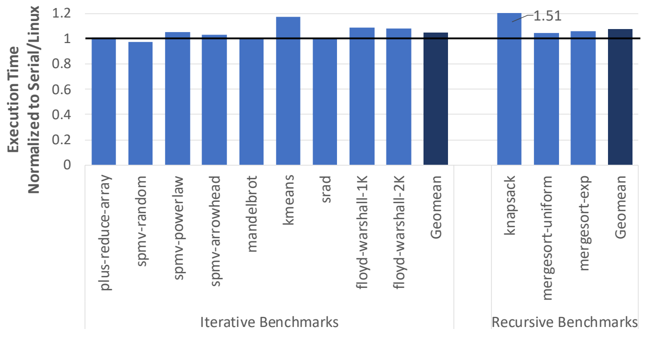


Figure 8. Normalized execution time of TPAL sans heartbeat interrupts and concomitant promotions, on Linux, single core. The compilation-related performance overhead of TPAL is minimal compared to sequential code.

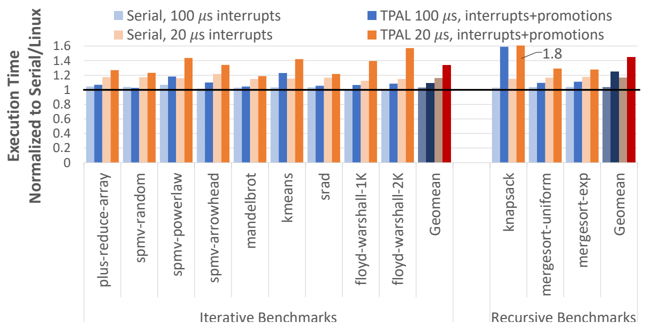


Figure 9. TPAL overheads including interrupts only, and interrupts plus promotions, on Linux, single core.

Interrupt-only overheads at ♡ = 100 μs are generally low, with a geomean of 3%. At ♡ = 20 μs, however, interrupt-only overheads approach 20% on several occasions, leading to a geomean as high as 16%. As we describe in more detail in Section 5.1, these overheads have a considerably higher and compounding effect at larger scales. However, it is possible to mitigate them through the direct use of timer and IPI hardware as enabled in Nautilus.

Promotion overhead is low but could be improved. Figure 9 also depicts the total overhead of promotions by taking the running times of our TPAL programs on a single core when allowing not only signals, but also promotions, to happen. At ♡ = 100 μs, all benchmarks except one input of *spmv*, *kmeans*, and *knapsack*, incur a low overhead at or below 11%. The overheads of *kmeans* and *knapsack* are already explained by the compilation-related overheads, which are not specific to TPAL. Although we cannot fully explain the cause of the 18% overhead of *spmv* with the powerlaw input, we have found that on our other test machines the overhead is closer to 10%. For ♡ = 20 μs, however, the overhead of promotions becomes unacceptable and reaches a geomean of 34%.

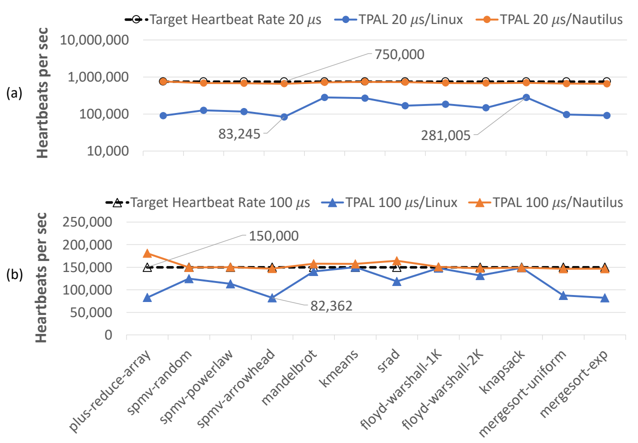


Figure 10. Achieved and target heartbeat rate in Linux and Nautilus, 15 cores.

Linux misses its target heartbeat rate. An even bigger problem in Linux is that, owing to high signaling overheads, it largely misses its target heartbeat rate. Figure 10 shows that Linux struggles to keep up with its target rate of 150K heartbeats/s across all 15 cores, even at a leisurely $\heartsuit = 100\mu s$. While it gets close to desired rate in 3 out of 12 benchmarks in our suite, in all other cases it misses its target, and most often by a lot. It is important to note that we present results for the best Linux mechanism (INT-PingThread); the INT-Papi mechanism performs even worse. The best Linux mechanism cannot even sustain heartbeat signals at a consistent rate for all benchmarks: for some it is as low as 82K/s, significantly lower than the desired rate of 150K/s.

We do not understand in detail why Linux signaling performs so poorly, but our results are in line with other observations of Linux behavior, as described earlier. Conceivably, a suitable Linux kernel mechanism could be designed to improve the situation, and this is a subject for future work.

The situation is aggravated for $\heartsuit = 20\mu s$ (Figure 10). Here Linux always misses its mark by a factor of 2.7–9×: while the target heartbeat rate is a rapid 750K heartbeats/s across all cores, Linux delivers at most 281K, and as low as 83K. Failure to achieve the target rate corresponds to potentially missed opportunities to extract parallelism from the application. We set out to mitigate this shortcoming by exploring alternate mechanisms to deliver interrupts reliably and at low cost.

Performance at scale. In Figure 11, we present speedup curves for all benchmarks. Overall, the curves suggest that TPAL and Cilk Plus achieve scaling as cores are added. However, the curves of TPAL usually show low overhead at small core counts, and highest performance at scale. There is one significant exception: *mandelbrot*. In *mandelbrot*, there is a need to spawn a large number of tasks, in order to keep the 15 cores fed, but the speedup curve of Cilk suggests that TPAL is not generating a sufficient number of tasks. The reason for insufficient tasks is that the signaling mechanism provided by Linux does not support a high enough throughput to meet the needs. In Nautilus, where the signaling mechanism shows better performance at scale, our *mandelbrot* benchmark scales very well, outperforming the Cilk Plus version.

5 OS Support for Heartbeat Scheduling

The signaling mechanisms available in Linux were not designed for the purpose of driving heartbeat interrupts at fine granularity, such as $\heartsuit = 20\mu s$—$100\mu s$. As shown shown by others [33], existing software mechanisms in Linux are unable to achieve predictably low latencies for out-of-band event signaling. While the performance of such signaling mechanisms ultimately depends on hardware capabilities, software overheads, a high degree of abstraction, and mismatched interfaces can introduce barriers to signaling performance. To understand the extent of these issues as they pertain to heartbeat signals, we implemented a prototype in a lightweight OS kernel framework called Nautilus that allows us to precisely control the software overheads of event signaling without significant effort.

5.1 Nautilus and the TPAL HRT

Nautilus is a lightweight kernel framework intended to support *hybrid runtime systems* (HRTs), i.e. language runtimes that have the full power of the OS kernel [34–36]. It is a publicly available open-source codebase[2] that currently runs directly on x64 NUMA hardware and Intel Xeon Phi, as well as in a unikernel configuration atop various virtualization platforms. Applications in Nautilus run in a single shared address space, in kernel-mode, with fully privileged access to the machine.

We created a prototype TPAL Hybrid Runtime (TPAL HRT) that runs in Nautilus. It is important to note that TPAL HRT uses application code that is identical to the Linux user-level implementation. The TPAL compiler transformations are no different. The TPAL runtime is also almost entirely identical. What is different is that everything runs within the kernel, and heartbeat signaling is accomplished directly using the x64 timer and interrupt hardware. Arguably, a representation that captures task parallelism, such as TPAL, makes such a radically different implementation feasible for the end-user.

Nautilus includes a lightweight, inter-core signaling framework called Nemo, which reduces signaling latency and jitter significantly [33]. Nemo is essentially a thin veneer around the standard hardware mechanisms for signaling between CPU cores, namely *inter-processor interrupts* (IPIs). In most architectures, IPIs represent the lower limit on architected, out-of-band event signaling. Their cost is typically within a few thousand cycles, most of which is consumed by interrupt handling overhead on the receive side CPU. OSes typically use IPIs for internal synchronization events, but Nautilus

[2] https://github.com/HExSA-Lab/nautilus

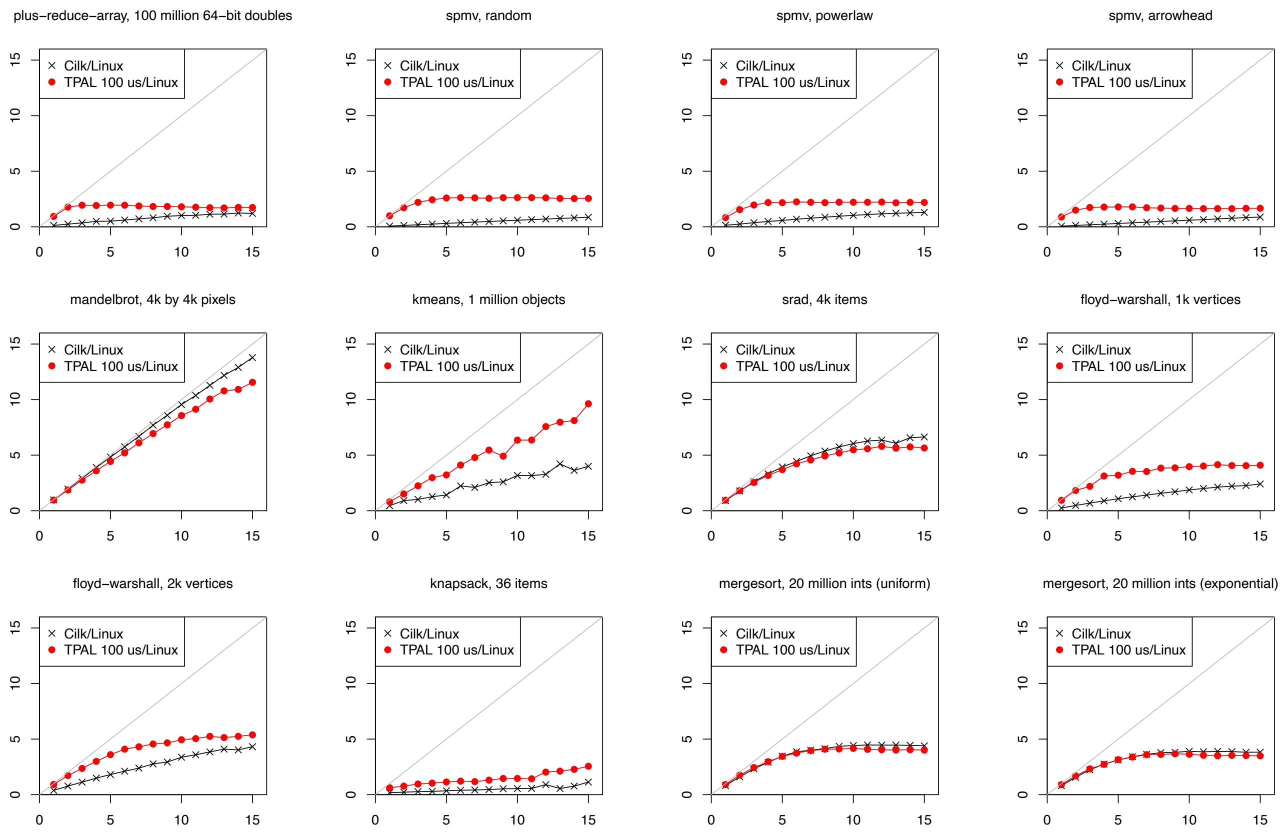


Figure 11. Speedup over Serial/Linux for Cilk Plus and TPAL/Linux, varying the number of cores.

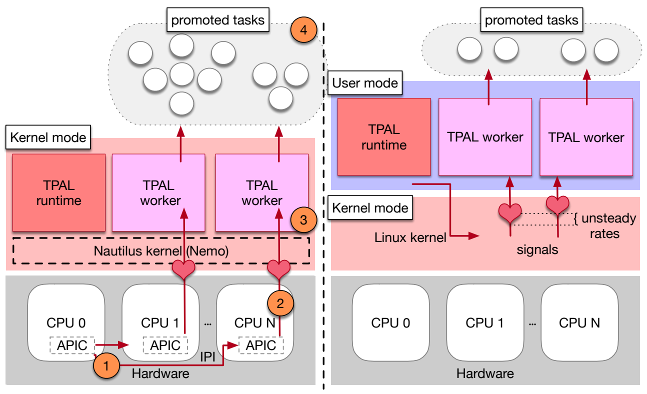


Figure 12. Heartbeat signaling mechanisms in Nautilus (left) and Linux (right).

exposes this capability to parallel runtimes through Nemo, allowing programmers to multiplex a fixed set of software events and their handler functions atop IPIs. Thus, Nemo is a natural fit for heartbeat signals.

We built our TPAL HRT directly atop Nemo and the hardware timer interrupts that Nautilus also exposes, as depicted in Figure 12. The first TPAL worker on CPU 0 registers a Nautilus timer handler that will be invoked at the specified heartbeat interval (♡)/rate. This builds upon the CPU's local APIC timer, which can (typically) be programmed to interrupt at intervals down to 10 ns. The timer interrupt directly triggers the heartbeat timer handler. The timer handler in turn uses Nemo to distribute a heartbeat signal to every other core, which Nemo does using IPIs (1). On the destinations, these IPIs trigger (2), via the Nemo framework, the TPAL workers (3), which have more opportunities to unleash parallelism in the form of task promotions (4) due to the consistently achieved heartbeat rate. This approach brings the limit on ♡ closer to the hardware limit.

5.2 Signaling Performance in TPAL HRT

Recall that Figure 9 showed the impact of the heartbeat signaling mechanism on Linux for 20 and 100 μs heartbeats. These single-core results were discussed in Section 4.4.

Figure 13 shows the corresponding results for TPAL HRT. The overheads for ♡ = 100 μs corresponding to signaling alone are completely masked, whereas in the best-case Linux implementation they were typically around 3–4% and as high as 7%. At 20 μs, while the Linux overheads are on the order of 13–22%, the TPAL HRT overheads are at most 4.9%, and are usually much lower. These gains cascade when promotions are also enabled. Clearly, it is Linux that imposes noticeable

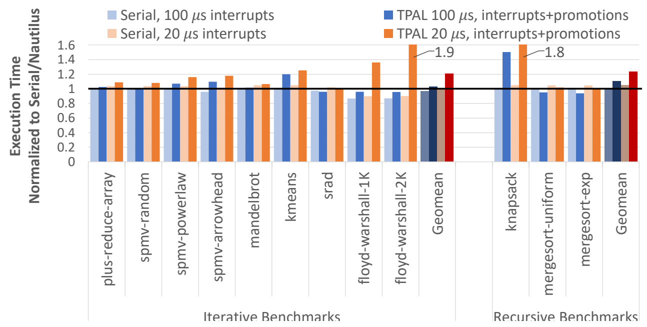


Figure 13. TPAL overheads including interrupts only, and interrupts plus promotions, on Nautilus, single core.

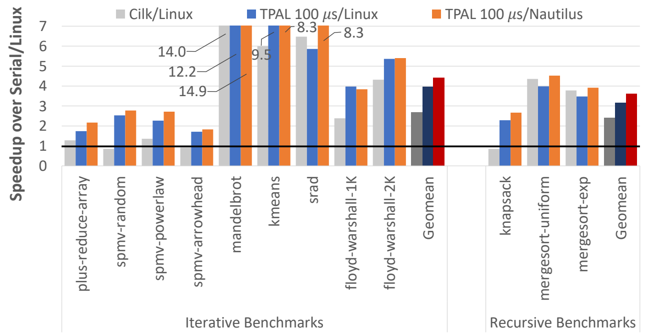


Figure 14. Speedups over serial execution on Linux of 15-core runs of Cilk Plus, TPAL/Linux and TPAL/Nautilus. TPAL outperforms serial/Linux and Cilk Plus on all benchmarks.

costs, even when ♡ is a leisurely 100 μs, as current hardware can indeed support lower-cost signaling.

As Nautilus is capable of exploiting the fast interrupt delivery mechanisms of modern hardware, it also achieves its target heartbeat rate. Figures 10 and 13 show that Nautilus practically always achieves the heartbeat rate that it is requested to deliver for both ♡ = 100 μs and ♡ = 20 μs. While the best Linux mechanism cannot even sustain heartbeat signals at a consistent rate for all benchmarks, even at ♡ = 100 μs, Nautilus not only hits the target, but it also delivers a much more consistent rate for both 100 μs and 20 μs. There is clearly a scaling issue that may affect performance in Linux at higher core counts, but does not affect Nautilus.

5.3 Putting It Together: Performance at Scale

Figure 14 compares the overall speedups achieved by Cilk and TPAL on Linux and Nautilus at scale (16 total cores) normalized over the serial execution on Linux. Here ♡ = 100 μs, which is generous to Linux.

Cilk Plus achieves speedups on two thirds of our workloads but incurs slowdowns on the remaining third. While some speedups are high (e.g., 14× for mandelbrot) often performance improvements are limited, leading to a respectable but less-than-desired speedup geomean of 1.9× for iterative benchmarks and 2.4× for recursive ones on 15 cores.

TPAL on Linux attains higher performance than Cilk Plus in most cases, or performs comparably well, leading to a geomean of 4× speedup for iterative benchmarks and 3.2× for recursive ones. Similarly, TPAL on Nautilus achieves speedup geomeans of 4.4× and 3.6× respectively for iterative and recursive workloads, outperforming both Cilk Plus and TPAL/Linux in aggregate. Looking at the individual workloads we observe that TPAL on Nautilus achieves the lowest wall-clock execution times than any other system (Cilk plus or TPAL/Linux) for all benchmarks except one: *kmeans*, in which TPAL/Linux narrowly beats TPAL/Nautilus by 12%. It appears that achieving the desired heartbeat interval/rate is a double-edged sword. On the one hand, TPAL HRT can significantly outperform the Linux implementations at scale—for example, in *srad* and *mandelbrot*. Here, the correct and stable heartbeats trigger useful promotions that manifest useful parallelism that contributes to performance, allowing, for example, TPAL to overcome the obstacles it was facing with *mandelbrot* on Linux, as noted in Section 4.3. On the other hand, when Linux fails to achieve the target heartbeat rate, this failure benefits benchmarks in which promotions are *not* desirable by the simple fact that there are fewer promotion opportunities due to this failure.

Overall, TPAL on Linux achieves significant speedups over Cilk (geomean 53%) for benchmarks that are amenable to recurrent decomposition, while the others (that do not lend well to our technique) incur minimal slowdown (geomean 9.8%). It is important to also note that if we consider both Linux and Nautilus implementations, TPAL strictly outperforms Cilk Plus: in all cases, at least one of our TPAL implementations achieves higher speedup than Cilk Plus, and more often both do.

6 Related Work

Many task-parallel programming languages have been developed, going back to the 1980s, including multiLisp [37], NESL [12, 14], Cilk (extending C) [30], several extensions of Java [17, 40, 42], parallel Haskell [45, 48, 54], several forms of parallel ML [10, 28, 32, 55, 57, 58, 61], and X10 [20]. All of these task-parallel languages rely on task-scheduling techniques that go back to Brent's seminal work [18], which has been extended in many directions [4, 8, 11, 13, 15, 16, 24, 43, 51]. These techniques primarily focus on reducing scheduling overheads of tasks that are already created. Task-creation overheads have also proved to be significant, and there has been work on reducing them [6, 25, 30, 38, 41, 60, 63], going back to Cilk-5's clone optimization. Our TPAL takes inspiration from prior work but takes a different tack: instead of reducing the cost of task creation—which can only be done up to a point—TPAL amortizes the cost against the abundant useful work that a program naturally performs.

There has been recent interest in improving the quality of code generated by high-level parallel languages, such as Cilk Plus. Tapir [56] extends LLVM to support Cilk Plus, bringing to parallel function calls and loops the benefits of LLVM's existing serial code-optimization passes. Although it addresses compiler optimization of parallel code, Tapir does not address granularity control, a major challenge in parallelizing codes efficiently, whereas TPAL does. Furthermore, by design, Tapir does not address compiler optimization in the stages following the lowering of Tapir's high-level parallel instructions, whereas TPAL does. We believe that there is now a feasible path to combine Tapir and TPAL, as we outlined in Section 3.1, which will feature the benefits of each approach in one system.

For our implementation of TPAL, we used the signaling mechanism provided by the OS, in our case, Linux, to drive heartbeats. The performance issues related to the Linux signal mechanism are relatively well known, and are explicitly noted in the source code. In particular, the perf sample rate is limited to 10usec for this reason [2] (lines 418 and 492). There is also some discussion of it in the research literature [62].

Alternatively, the heartbeats can be driven by *software polling* [26]. In software polling, there is typically some sophisticated compiler support that inserts into programs the branch instructions needed to drive heartbeats. This approach has some advantages, especially in the context of managed languages, where there may be preexisting support for software polling. However, there are two challenges facing any implementation of software polling. First, it can block compiler optimizations if not implemented carefully. Second, an implementation needs to ensure there is enough space between polling events to keep polling overheads low, but close enough to consistently meet the target hartbeat rate. These challenges have been addressed by advanced Java runtimes, where there is evidence showing that the overhead cost of the polling is close to 2% [47]. To achieve this result, it is crucial to use a single load/cmp/branch sequence to ensure that the branch would staticalliy be predicted as not taken, and that the register allocation was not affected by the unlikely branch and call [1, 9]. Also, in non-managed languages, such as C++, it can be appropriate to use software polling, and there has been work on bringing compiler support to LLVM [31]. We plan to experiment with such alternative mechanisms in future implementations of TPAL.

7 Conclusion

When it comes to performance, it takes two to tango: parallel and sequential computation. Today, we expect the programmer to choreograph carefully, when exactly each will take a step. If parallel goes too far, performance will suffer because of the overheads associated with realizing it in practice, such as task creation, scheduling, etc. If sequential goes too far, scalability will suffer. This choreography involves difficult compromises and requires carefully optimizing code to make sure that each gets their "fair" share by considering everything from lower level concerns such as architectural constant factors, to compiler optimizations, and finally to algorithms.

With widespread availability of multicore hardware, we want languages that encourage and enable writing high-level parallel programs without all of these compromises. To this end, we proposed TPAL as a foundation for writing task-parallel programs. TPAL delivers the right level of parallelism by construction, always and consistently, and applies to irregular, nested, and loop-based parallel codes. We implemented and evaluated TPAL, considering important implementation tradeoffs at the runtime/OS level, on a challenging suite of benchmarks, featuring irregular, fine-grain parallelism. We showed that TPAL achieves consistent efficiency and an ability to find the right amount of parallelism regardless of workload.

Acknowledgments

U. Acar and M. Rainey. This work was partially supported by the National Science Foundation under grant numbers CCF-1901381 and 2028921.

R. Newton. This work was made possible by support from the National Science Foundation via awards CCF-2127277 and SPX-1725679.

P. Dinda and N. Hardavellas. This work was made possible by support from the National Science Foundation via awards CCF-1533560, CNS-1763743, and CCF-2028851.

S. Campanoni. This work was made possible by support from the National Science Foundation via award CCF-1908488 (also, CNS-1763743, and CCF-2028851).

K. Hale. This work was made possible by support from the National Science Foundation via awards CNS-1763612, CNS-1718252, and CCF-2028958.

References

[1] [n.d.]. Architecture and Policy for Adaptive Optimization in Virtual Machines. https://researcher.watson.ibm.com/researcher/files/us-groved/RC23429.pdf. Accessed: 2021-04-1.

[2] [n.d.]. The Linux source code file core.c. https://github.com/torvalds/linux/blob/master/kernel/events/core.c. Accessed: 2021-04-1.

[3] Umut A. Acar, Vitaly Aksenov, Arthur Charguéraud, and Mike Rainey. 2019. Provably and Practically Efficient Granularity Control. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. 214–228.

[4] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2002. The data locality of work stealing. *Theory of Computing Systems (TOCS)* 35, 3 (2002), 321–347.

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

[6] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2013. Scheduling Parallel Programs by Work Stealing with Private Deques. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*.

[7] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2016. Oracle-guided scheduling for controlling granularity in implicitly parallel languages. *Journal of Functional Programming (JFP)* 26 (2016), e23.

[8] Shivali Agarwal, Rajkishore Barik, Dan Bonachea, Vivek Sarkar, R. K. Shyamasundar, and Katherine A. Yelick. 2007. Deadlock-free scheduling of X10 computations with bounded resources. In *SPAA 2007: Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Diego, California, USA, June 9-11, 2007*. 229–240.

[9] Matthew Arnold and David Grove. 2005. Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '05)*. IEEE Computer Society, USA, 51–62. https://doi.org/10.1109/CGO.2005.9

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

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

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

[13] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. 1999. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM* 46 (March 1999), 281–321. Issue 2.

[14] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. 1994. Implementation of a Portable Nested Data-Parallel Language. *J. Parallel Distrib. Comput.* 21, 1 (1994), 4–14.

[15] Robert D. Blumofe and Charles E. Leiserson. 1998. Space-Efficient Scheduling of Multithreaded Computations. *SIAM J. Comput.* 27, 1 (1998), 202–229.

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

[17] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications* (Orlando, Florida, USA) *(OOPSLA '09)*. 97–116.

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

[19] F. Warren Burton and M. Ronan Sleep. 1981. Executing functional programs on a virtual tree of processors. In *Functional Programming Languages and Computer Architecture (FPCA '81)*. ACM Press, 187–194.

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

[21] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC) (IISWC '09)*. IEEE Computer Society, USA, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[22] Intel Corporation. 2014. *Intel C++ Compiler Code Samples*. https://software.intel.com/en-us/code-samples/intel-compiler/intel-compiler-features/IntelCilkPlus

[23] Daniele Cono D'Elia and Camil Demetrescu. 2018. On-stack replacement, distilled. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 166–180.

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

[25] Karl-Filip Faxén. 2009. Wool-A Work Stealing Library. *SIGARCH Comput. Archit. News* 36, 5 (June 2009), 93–100. https://doi.org/10.1145/1556444.1556457

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

[27] Stephen J Fink and Feng Qian. 2003. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003*. IEEE, 241–252.

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

[29] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. 2009. Reducers and Other Cilk++ Hyperobjects. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures* (Calgary, AB, Canada) *(SPAA '09)*. Association for Computing Machinery, New York, NY, USA, 79–90. https://doi.org/10.1145/1583991.1584017

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

[31] Souradip Ghosh, Michael Cuevas, Simone Campanoni, and Peter Dinda. 2020. Compiler-based timing for extremely fine-grain preemptive parallelism. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 736–750.

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

[33] Kyle Hale and Peter Dinda. 2018. An Evaluation of Asynchronous Software Events on Modern Hardware. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 355–368.

[34] Kyle C. Hale and Peter A. Dinda. 2015. A Case for Transforming Parallel Runtimes Into Operating System Kernels. In *Proceedings of the 24^{th} ACM Symposium on High-performance Parallel and Distributed Computing (HPDC '15)*.

[35] Kyle C. Hale and Peter A. Dinda. 2016. Enabling Hybrid Parallel Runtimes Through Kernel and Virtualization Support. In *Proceedings of the 12^{th} ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'16)*. 161–175.

[36] Kyle C. Hale, Conor Hetland, and Peter A. Dinda. 2016. Automatic Hybridization of Runtime Systems. In *Proceedings of the 25^{th} ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC '16)*. 137–140.

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

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

[39] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*. 32–43.

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

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

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

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

[44] Charles Leiserson and Aske Plaat. 1998. Programming parallel applications in Cilk. *SINEWS: SIAM News* 31, 4 (1998), 6–7.

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

[46] Vasileios Liaskovitis, Shimin Chen, Phillip B. Gibbons, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Michael Kozuch, Todd C. Mowry, and Chris Wilkerson. 2006. Parallel Depth First vs. Work Stealing Schedulers on CMP Architectures. In *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Cambridge, Massachusetts, USA) *(SPAA '06)*. Association for Computing Machinery, New York, NY, USA, 330. https://doi.org/10.1145/1148109.1148167

[47] Yi Lin, Kunshan Wang, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. 2015. Stop and Go: Understanding Yieldpoint Behavior. In *Proceedings of the 2015 International Symposium on Memory Management* (Portland, OR, USA) *(ISMM '15)*. Association for Computing Machinery, New York, NY, USA, 70–80. https://doi.org/10.1145/2754169.2754187

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

[49] David Mosberger, Peter Druschel, and Larry L Peterson. 1996. Implementing atomic sequences on uniprocessors using rollforward. *Software: Practice and Experience* 26, 1 (1996), 1–23.

[50] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. 1999. PAPI: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, Vol. 710.

[51] Girija J. Narlikar and Guy E. Blelloch. 1999. Space-Efficient Scheduling of Nested Parallelism. *ACM Transactions on Programming Languages and Systems* 21 (1999).

[52] MEJ Newman. 2005. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics* 46, 5 (Sep 2005), 323–351. https://doi.org/10.1080/00107510500052444

[53] OpenMP Architecture Review Board. [n.d.]. OpenMP Application Program Interface. http://www.openmp.org/

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

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

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

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

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

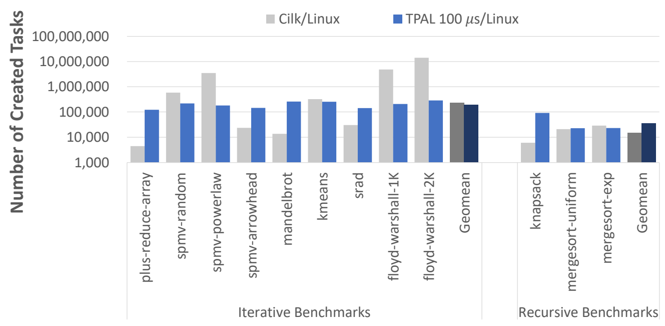
[59] Torbjørn Johnsen Tessem. 2013. *Improving parallel sparse matrix-vector multiplication*. Master's thesis. The University of Bergen.

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

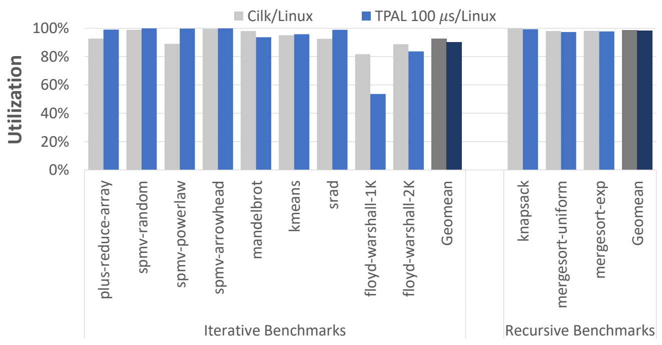
[61] Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. 2020. Disentanglement in Nested-Parallel Programs. In *Proceedings of the 47th Annual ACM Symposium on Principles of Programming Languages (POPL)"*.

[62] Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. 2015. Computer Performance Microscopy with Shim. *SIGARCH Comput. Archit. News* 43, 3S (June 2015), 170–184. https://doi.org/10.1145/2872887.2750401

[63] Christopher S Zakian, Timothy AK Zakian, Abhishek Kulkarni, Buddhika Chamith, and Ryan R Newton. 2015. Concurrent Cilk: Lazy Promotion from Tasks to Threads in C/C++. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer International Publishing, 73–90.



(a) Number of tasks (promotions) generated by each technique.



(b) Utilization (i.e., fraction of execution time spent on useful work).

Figure 15. The number of generated tasks and utilization for Cilk Plus and TPAL on Linux, 15 cores.

A Additional Experimental Results

Figures 15a and 15b present the number of tasks generated by Cilk and TPAL when running on Linux at ♡ = 100 μs, along with the utilization that each technique achieves across our benchmark suite, respectively.

Figure 15a shows that TPAL creates more tasks than Cilk Plus for about half the benchmarks in our suite, while for the remaining half it does not. Yet, as Figure 7 indicates, TPAL outperforms Cilk Plus at scale. This seeming discrepancy highlights the ability of recurrent decomposition to find the right balance between the amount of extracted parallelism and task size. Too many and too small tasks will inevitably accumulate overhead on the program's execution and may outweigh some of the benefits of parallel execution.

Figure 15b shows that TPAL generally achieves comparable or higher utilization than Cilk Plus across our benchmark suite, with the exception of four benchmarks, most notable of which is *floyd-warshall* with 1K inputs. While TPAL's utilization is much lower than Cilk Plus' for this benchmark, TPAL outperforms Cilk Plus because it creates just enough tasks to keep the cores fed with useful work, but not wasting time sharing with tasks that are too small. This behavior was discussed at length in Section 4.3.

B Nested Parallelism

B.1 Loop-based nested parallelism

In TPAL, it is possible to write programs that nest loops in any desired fashion. There may be parallel loops inside serial ones, or vice versa, or parallel loops inside other parallel ones. For example, we can write a parallel pow program that computes the result $\mathsf{f} = \mathsf{d}^{\mathsf{e}}$ by executing our prod program inside an outer, pow loop. The steps of writing such a pow program are similar to those we outlined for prod: we need serial blocks, handler blocks, and parallel blocks. Each time around the pow loop, there is going to be a jump to prod program. Symmetrically, there must a replacement of the instruction at line 7 of Figure 2 to jump back to the pow loop. Recall from before that by default our prod program starts executing in a serial fashion. This property is crucial for performance, especially in loop nests. By starting the inner loop serial first, we get to best of both worlds: if the inner loop turns out to be short lived, we pay zero overhead cost for parallelism, but if it is sufficiently long lived, we can access its parallelism.

To complete the implementation, there is one additional change we need to make to our prod program. In Heartbeat Scheduling, a promotion must always manifest the *outer-most* latent parallelism in the program. To satisfy this constraint in our pow program, we must always promote from remaining iterations of the outer (i.e., pow) loop if possible (i.e., there are at least two or more remaining). Otherwise, if there is no such latent outer parallelism, we can try to promote from inner parallelism (i.e., in the prod loop). In order to implement this policy, we have to modify the handler blocks of our prod loop: the modified blocks must try to promote from remaining iterations of the pow loop first, and only if that attempt fails, try to promote from remaining iterations of the prod loop.

To see how this policy is implemented, let us build on our running prod example by nesting that loop-based program to function as the inner loop in TPAL version of the pow program in Figure 16. In particular, our pow program computes $\mathsf{f} = \mathsf{d}^{\mathsf{e}}$ by iterating multiplications of an accumulator register pr by d, using prod to perform the multiplications. In Figure 17, we see there are four blocks that handle the initial sequential execution of pow. Just like with prod, the loop and exit blocks have promotion-ready program point and join-target annotations. In this case, these annotations enable the outer parallelism for the pow loop. Unlike prod, there is a continuation block `ploop-cont`, which is the return continuation of the inner loop.

```
// computes f = d^e
reducer<op_mul<int>> pr(1)
cilk_for(int j = 0; j != e; j++)
  reducer<op_add<int>> r(0)
  cilk_for(int i = 0; i != d; i++)
    r += *pr
  pr *= *r
f := *pr
```

Figure 16. The pow program

```
pow: [·] // computes f = d^e
  pr := 1
  pjr := 0
  jump ploop

pexit: [jtppt assoc-comm;
                {pr ↦ pr2}; pcomb]
  f := pr
  jump pret

ploop: [prppt ptry-promote]
  if-jump e, pexit
  a := d
  b := pr
  ret := ploop-cont
  jump prod

ploop-cont: [·]
  pr := c
  e := e - 1
  jump ploop
```

Figure 17. The sequential blocks of the pow program.

The part of the program that implements the promote-the-outermost-parallelism policy of heartbeat scheduling is shown in Figure 18. These handler blocks implement the following behavior: if it is the case that, from the pow loop, $d \geq 2$, then our handler promotes a new parallel task for the outer, pow loop. If, instead it is the case that, $d < 2$ but $a \geq 2$, our handler promotes a new parallel task for the inner (i.e., prod) loop. Otherwise, the handler returns control to whichever block was interrupted. To implement this overall strategy, we need to make just two modifications to prod: we have to change the promotion-ready program point annotations to instead point to our new handler block for pow, namely ptry-promote. The remaining blocks of pow are the ones shown in Figure 19, which implement the parallel part of the program.

B.2 Recursive parallelism

Given a minor extension, TPAL can support recursive parallelism, e.g., in the style of Cilk spawn/sync syntax, along with ordinary, serial function calls and stack and heap memory. The extension brings support for parallel and serial function calls by introducing call stacks, which are managed by TPAL programs. TPAL is agnostic to stack representation: it can use the usual linear C representation or, e.g., a segmented one. It is also agnostic to calling convention: it can support the usual conventions of RISC architectures and x64.

In TPAL, the convention for issuing parallel calls uses the approach proposed for Heartbeat Scheduling. Each task has a private call stack and issues parallel and serial calls in much the same way, by pushing and popping frames from the end of the call stack. The difference is parallel calls require some additional bookeeping in the call stack of each task: the *promotion-ready mark list*. The promotion-ready mark list consists of a linked list stored across stack frames, such that each node in the list points into a frame. These pointers exist to be accessed by promotion handlers, whose purpose is to try to manifest latent parallelism held in some state stored in a frame. If it holds onto some latent parallelism, before it makes a call to some other function, a function must "advertise" that latent parallelism by registering a mark in its own frame. Symmetrically, when it eliminates latent parallelism, a function must remove the corresponding mark. TPAL provides three instructions for managing the marks: two for pushing and popping marks in the current frame of a task, and one for accessing the least-recent mark in the current task.

Access to the least-recent mark is essential, because of the outer-most policy of Heartbeat Scheduling. To support this policy, each handler in the program must first try to promote any parallelism in the current call stack before trying to promote any loop-level parallelism. Moreover, any such handler must promote parallelism from the least-recent frame in the mark list. The overhead carried by our parallel calling convention is small: it requires only modest support beyond existing loop-based parallelism, in the common case requiring maintenance of just a low-cost mark structure, and supports all combinations of nesting with parallel and serial loops.

```
ptry-promote: [·]
  pabort := ploop
  ploop-promote-cont := ploop-par
  if-jump pjr, ploop-try-promote
  pabort := ploop-par
  jump ploop-par-try-promote

loop-try-promote: [·]
  pabort := loop-try-promote
  ploop-promote-cont := loop
  if-jump pjr, ploop-try-promote
  jump ploop-par-try-promote

loop-par-try-promote: [·]
  pabort := loop-par-try-promote
  ploop-promote-cont := loop-par
  if-jump pjr, ploop-try-promote
  jump ploop-par-try-promote

ploop-try-promote: [·]
  t := e < 2
  if-jump t, pabort
  pjr := jralloc pexit
  jump ploop-promote

ploop-par-try-promote: [·]
  t := e < 2
  if-jump t, pabort
  jump ploop-promote

ploop-promote: [·]
  m := e / 2
  n := e % 2
  e := m
  tr := pr
  pr := 1
  // ↓ needed for prod
  ret := ploop-par-cont
  fork pjr, ploop-par
  e := m + n
  pr := tr
  jump ploop-promote-cont
```

Figure 18. Promotion-handler blocks of the pow program.

```
pcomb: [·]
  pr := pr * pr2
  join pjr

ploop-par: [prppt ptry-promote]
  if-jump e pjoin
  a := d
  b := pr
  ret := ploop-par-cont
  jump prod

ploop-par-cont: [·]
  pr := c
  e := e - 1
  jump ploop-par

pjoin: [·]
  join pjr
```

Figure 19. The parallel blocks of the pow program.

To enable recursive parallelism, we extend the grammar of TPAL, as shown in Figure 21, with support for stack-based memory. Support for heap-based memory, e.g., by introducing malloc/free, is also possible, but we omit it to simplify the presentation. The extensions to TPAL that are relevant for our present purpose are the eight new instructions in the grammar. The first four are conventional: we provide instructions for allocating and deallocating a given number of words from a stack, for which the stack pointer is given in a register, and we provide load and store instructions that can access memory at an given address. The addressing mode is specified by a base pointer register, plus an integer-literal offset. The four remaining instructions assist in the promotion of tasks that use stack memory, e.g., to make function calls.

In heartbeat scheduling, promotion assigns highest priority to latent parallelism in the outermost execution context, much like in the classic work-stealing algorithm. As such, to implement promotion efficiently for recursive programs, such as our Cilk-based fib program, the TPAL stack needs to provide constant-time access to the outermost promotable frame. Once this frame is promoted, we need access to the next one, and so on. Thus, at a minimum, we need a singly linked list between promotable frames from top to bottom, i.e., staring from the oldest frames. Yet, at the same time, the execution of a task pushes and pops frames at the bottom of its stack. In particular, it is possible that a promotable frame gets popped before it is promoted (e.g., the left branch of a pair terminates before the right branch gets promoted). Efficiently removing the frame from the singly linked list between promotable frames requires reverse pointers, hence the need for a doubly linked list. To represent this

```
int fib(int n)
  if n < 2
    return n
  int f1 =
    cilk_spawn fib(n - 1)
  int f2 = fib(n - 2)
  cilk_sync
  return f1 + f2
```

Figure 20. The `fib` program

$$
\begin{array}{llll}
\text{instructions} & \imath & ::= & \ldots \\
 & & \mid & r \ := \ \mathbf{snew} \\
 & & \mid & \mathbf{salloc}\, r, n \mid \mathbf{sfree}\, r, n \\
 & & \mid & r \ := \ \mathbf{mem}[r + n] \mid \mathbf{mem}[r + n] \ := \ v \\
 & & \mid & \mathbf{prmpush}\, \mathbf{mem}[r + n] \mid \mathbf{prmpop}\, \mathbf{mem}[r + n] \\
 & & \mid & r := \mathbf{prmempty}\, r \mid \mathbf{prmsplit}\, r, rr
\end{array}
$$

Figure 21. Extensions for TPAL to support stack memory

doubly linked list, promotable frames include a prev and a next pointer, using null to terminate the list. We call this linked list the *promotion-ready mark list.*

We demonstrate our promotion-aware stack data structure with an example implementation of the recursive `fib` function in TPAL. The sequential blocks of the program are given in Figure 22, and the parallel blocks in Figure 23. We give a sample trace of this program in Figure 24, where we step through the first two recursive calls of the application `fib(3)`. In the first step, we have the initial frame, which is created by the `fib` block. In the second, the program pushes a new frame for the continuation of the first recursive call, `fib(2)`, and in the third, a frame corresponding to the continuation of `fib(1)`. In the process of creating these frames, the `loop` block pushes promotion-ready marks as well, using the instruction at line 19. In the resulting trace step, there are two promotion-ready marks in the stack, each representing instances of latent parallelism that exists between the first branch (i.e., the call `fib(n - 1)`) and its continuation, the second branch (i.e., the call `fib(n - 2)`).

Now, suppose that, by the next step, a heartbeat interrupt triggers the handler for the `loop` block, and execution continues through to line 46 in the handler. This sequence of instructions has to perform the usual setup for a promotion: create a new join record, and prepare registers for the fork instruction. It also has to modify the stack so that the continuation of the current task and the child task each terminate by issuing a join instruction, which satisfies a dependency on the join continuation task. The stack of the running task is modified, first by the promotion mark-list split instruction on line 44, and then by the write issued on line 46, where the resulting stack corresponds to the last one in our trace. The split operation pops the oldest promotion-ready mark from the stack (replacing it by a nil value), and assigns `sp-top` to point at the cell one past the old promotion mark. The write issued on line 46 overwrites the original continuation targeting `branch1` with a new continuation, which targets the `joink` block.

After the fork instruction executes, we have a program state corresponding to the diagram in Figure 25. The interrupted computation continues to run in its original task, T_1, and the forked branch in the task T_2. Each of these new tasks has a dependency the join continuation T_j. In this case, the join continuation is trivial, because it just jumps to the `exit` block, which is the termination point of the whole `fib` program. But in general, there may be an arbitrary join continuation in its place, if `fib` is called by another function. As a result of the promotion, the task T_j inherits the segment of the stack below the frame that is promoted. Our semantics is prescriptive only for the high-level behavior of the stack, not to its implementation: it may involve copying out the frames for the join continuation or allowing regions of the stack to be divided among parent and child tasks. In the implementation section, we detail the various implementation strategies for a stack.

```
// computes f =  fib( n)
fib: [·]
  salloc sp, 1
  mem[sp + 0] := exit
  jump loop

exit: [·]
  sfree sp, 1
  jump ret

loop: [prppt loop-try-promote]
  f := n
  t := n < 2
  if-jump t, retk
  f := 0
  salloc sp, 3
  mem[sp + 0] := branch1
  t := n - 2
  prmpush mem[sp + 1]
  mem[sp + 2] := t
  n := n - 1
  jump loop
```

```
retk: [jtppt assoc-comm;
            {f ↦ f2}; comb]
  t := mem[sp + 0]
  jump t

branch1 : [·]
  mem[sp + 0] := branch2
  prmpop mem[sp + 1]
  n := mem[sp + 2]
  mem[sp + 2] := f
  jump loop

branch2 : [·]
  t := mem[sp + 2]
  f := f + t
  sfree sp, 3
  jump retk
```

Figure 22. The sequential blocks of the `fib` program.

```
loop-try-promote: [·]
  t := prmempty sp
  if-jump t, loop
  jr := jralloc retk
  prmsplit sp, top
  sp-top := sp + top - 1
  mem[sp + 0] := joink
  tn := n
  n := mem[sp-top + 2]
  tsp := sp
  sp := snew
  salloc sp, 1
  mem[sp + 0] := joink
  fork jr, loop-par
  sp := tsp
  n := tn
  jump loop-par
```

```
comb: [·]
  f := f + f2
  join jr

joink: [·]
  sp := sp-top + 3
  join jr

loop-par: [prppt loop-par-try-promote]
   // similar to the loop block
   ...

loop-par-try-promote: [·]
  // similar to the loop-try-promote
    block
  ...
```

Figure 23. The parallel blocks of the `fib` program.

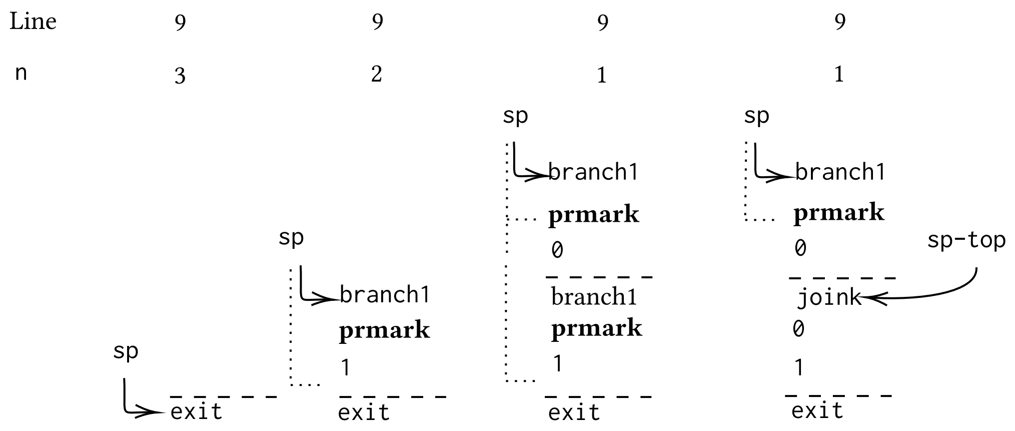


Figure 24. Trace of the stack contents of executing `fib(3)`. Each stack in the diagram represents the contents of the stack just after executing the line of `fib` listed above. The dashed lines mark the boundary between two stack frames. The dotted lines represent the linking between promotion-ready marks in the stack.

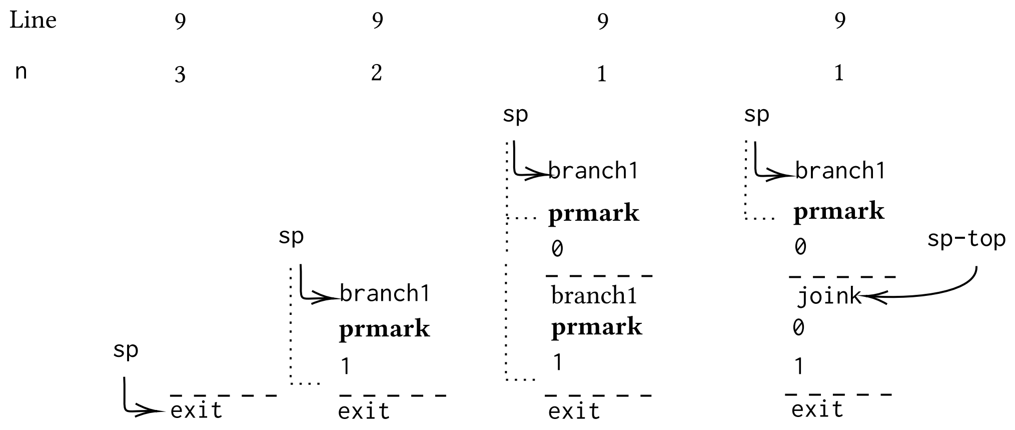


Figure 25. Result of promiting `fib(3)`

$$\begin{array}{llcl}
\text{register files} & R & ::= & \{\, r_1 \mapsto v_1,\ \ldots\ , r_n \mapsto v_n \,\} \\
\text{heaps} & H & ::= & \{\, l_1 \mapsto h_1,\ \ldots\ , l_n \mapsto h_n \,\} \\
\text{heap values} & h & ::= & B \\
\text{program counters} & \bar{l} & ::= & l_{[\mathbb{N}]} \\
\text{cycle counters} & \diamond & \in & \mathbb{N} \\
\text{tasks} & T & ::= & \langle \bar{l}; \diamond; H; R; I \rangle
\end{array}$$

$$\begin{array}{llcl}
\text{join-activation-record maps} & J & ::= & \{\, j_1 \mapsto jr_1,\ \ldots\ , j_1 \mapsto jr_n \,\} \\
\text{join-activation records} & jr & ::= & (l; js) \\
\text{join statuses} & js & ::= & \textbf{jsopen} \mid \textbf{jsclosed}
\end{array}$$

$$\begin{array}{llcl}
\multicolumn{4}{c}{\text{extensions for stacks:}} \\
\text{heap values} & h & ::= & \ldots \mid \textbf{tup}\ (v_1,\ \ldots\ , v_n) \\
\text{operands} & v & ::= & \ldots \\
 & & \mid & \textbf{uptr}\ h \mid \textbf{prmark}
\end{array}$$

Figure 26. Formal grammar of the TPAL evaluator.

$$\begin{array}{lcl}
\hat{R}(r) & = & R(r) \\
\hat{R}(n) & = & n \\
\hat{R}(l) & = & l
\end{array}$$

$$\begin{array}{lcl}
\hat{H}(R, v) & = & (l_t, B_t) \\
 & & \text{where} \quad l_t = \hat{R}(v) \\
 & & \phantom{\text{where}} \quad B_t = H(l_t)
\end{array}$$

$$\begin{array}{lcl}
\text{PromotionReady}(l_{[n]}, H, \diamond) & = & (n \equiv 0) \wedge (H(l) = [\textbf{prppt}\, l']\ I) \wedge (\diamond > \heartsuit) \\[1em]
\text{MergeJ}(J_1, J_2) & = & J_1 \cup \{\, j \mapsto jr \mid (j \mapsto jr) \in J_2 \wedge j \notin dom(J_1) \,\} \\[1em]
\text{MergeR}(R_1, R_2, \Delta R) & = & \{\, r \mapsto v \mid (r \mapsto v) \in R_1 \wedge r \notin dom(\Delta R) \,\} \\
 & \cup & \{\, r_t \mapsto v \mid (r_s \mapsto v) \in R_2, (r_s \mapsto r_t) \in \Delta R \,\} \\[1em]
\text{MergeH}(H_1, H_2) & = & H_1 \cup \{\, l \mapsto h \mid (l \mapsto h) \in H_2 \wedge l \notin dom(H_1) \,\}
\end{array}$$

Figure 27. TPAL metafunctions.

C Formal model

We present the grammar of our TPAL evaluator in Figure 26. A register file R is a mapping from registers to values. A heap H is a mapping from labels to heap values. For the semantics with register-based memory only, a heap value h can be only a source block. The extension for call stacks adds tuples, which are the underlying representation used for stacks. A program counter $\bar{l}$ is a label paired with a number. The number represents an offset from the first instruction of the block associated with the label. A cycle counter $\diamond$ is a number that counts the number of elapsed instructions since the previous heartbeat interrupt. A task T is a tuple consisting of a program counter, a cycle counter, a heap, a register file, and a current instruction sequence. A join-activation-record map J is a mapping from join-record ids to join records. A join-activation record jr is a pair consisting of a continuation label and a join status, which is either open or closed.

In Figure 27, we present the metafunctions used by the TPAL evaluator. The promotion-ready metafunction checks for the need to service a heartbeat interrupt by inspecting the program counter and cycle counter. The merge operations handle merging for the respective environments of two tasks meeting at a join point. Merging of join records and heaps is straightforward because there can be no conflicts: tasks can only increase the overall information content in these environments. Merging of register files is a little more complicated. The contents of the result register file consist of a copy of the contents of the first register file, along with selected contents from the second. The selected contents consist of only those registers mentioned in the register-renaming environment.

$$\text{cost graphs} \quad g \quad ::= \quad \mathbf{0} \mid \mathbf{1} \mid (g \cdot g) \mid (g \parallel g)$$
$$\text{cost of task creation (in nb. of cycles) } \tau$$

$$\begin{array}{lcl}
\text{Work}(\mathbf{0}) & = & 0 \\
\text{Work}(\mathbf{1}) & = & 1 \\
\text{Work}(g_1 \cdot g_2) & = & \text{Work}(g_1) + \text{Work}(g_2) \\
\text{Work}(g_1 \parallel g_2) & = & \tau + \text{Work}(g_1) + \text{Work}(g_2) \\
\\
\text{Span}(\mathbf{0}) & = & 0 \\
\text{Span}(\mathbf{1}) & = & 1 \\
\text{Span}(g_1 \cdot g_2) & = & \text{Span}(g_1) + \text{Span}(g_2) \\
\text{Span}(g_1 \parallel g_2) & = & \tau + \max(\text{Span}(g_1), \text{Span}(g_2))
\end{array}$$

Figure 28. Cost semantics of TPAL

We define the components needed for the cost model in Figure 28. We use cost graphs as a convenient way to formalize the work and span of an execution. The execution of a TPAL program induces a series-parallel, directed acyclic graph. The grammar of cost graph includes: the empty graph, written **0**, the one-vertex graph, written **1**, sequential composition of two graphs, written $(g_1 \cdot g_2)$, and parallel composition of two graphs, written $(g_1 \parallel g_2)$. Figure Figure 28 also gives the formal definition of the work and span of cost graph g, written Work(g) and Span(g), respectively. We weight fork-join operations with some cost τ. This fixed parameter τ represents the runtime overhead associated with a fork-join operation.

We show the transition rules for single-task steps in Figure 29. A step takes a task record to a changed task record, or to a halt state.

$$(\bar{l}, H, R, I) \;\rightarrow\; (\bar{l}', H', R', I')$$

The move transition assigns the contents of a register. The binary op transition performs a given binary operation and assigns the result to a register. The conditional branch transitions handle both cases of an if instruction. We interpret the number zero to represent a true value and all others false. An unconditional branch jumps to a specified label. Finally, a halt instruction leaves the task record unchanged.

In Figure 30, we present the transitions for a multi-task evaluation. For simplicity, we use a big-step semantics. The judgment takes a join-record map J and a task T and evaluates the task to a final configuration T', a final join-record map J', and a cost graph g.

$$J; T \;\Downarrow\; J'; T'; g$$

The sequential transition applies when the task is ready to perform a serial transition step. The join-record allocation transition allocates for the task a fresh join-record in the join-record map. Initially, this join record is in a close state, meaning that there are one or zero tasks with dependency edges registered on the join record. The fork rule generates subderivations for parent and child tasks, joins their results to generate a subderivation for the combine task, and delivers the result of the combine task as its final result. The join-block transition applies when parent and child tasks issue a join instruction. The join-continue transition applies when the combine block issues a join instruction. At this point, the issuing task proceeds to jump to the join continuation block stored in the join record, thereby completing the parallel loop. Finally, the try-promote transition fires only when a promotion is ready. The guard on all the rules ensure that the try-promote rule is mutually exclusive with all others.

C.1 Support for stack memory

In Figure 31, we show the transition rules for handling stack instructions. The first transition rule allocates a fresh stack to the caller and stores a pointer in the destination register. The second and third allocate and free a specified number of cells in the stack pointed to by the first register operand. The stack store transition writes a value in a specified operand to a stack cell, specified by a register holding a stack pointer and an integer offset. The stack load transition reads a value from a stack cell, as specified by a register holding the stack pointer and an integer literal offset. The promotion-mark empty transition rules write a zero (true) in a destination register if the promotion-ready mark list is empty, and nonzero (false) otherwise. The promotion-ready mark push and pop transitions push and pop a mark at a cell in the stack, specified by a register holding a stack pointer and an integer literal offset. The promotion-ready mark split transition pops the outermost (i.e., least recent) promotion-ready mark in the stack. Its first operand specifies a pointer to the call stack and its second a destination register in

[MOVE]

$$\frac{R[r = \hat{R}(v)] = r'}{(l_{[n]}, H, R, r \; := \; v; I) \;\longrightarrow\; (l_{[n+1]}, H, R', I)}$$

[BINOP]

$$\frac{\begin{array}{c} R(r) = n_1 \quad \hat{R}(v) = n_2 \\ R[r_d = op(n_1, n_2)] = R' \end{array}}{(l_{[n]}, H, R, r_d \; := \; op \; v, r; I) \;\longrightarrow\; (l_{[n+1]}, H, R', I)}$$

[IF-TRUE]

$$\frac{\hat{H}(R, v) = (l', [\star] \; I') \quad R(r) = 0}{(l_{[n]}, H, R, \textbf{if-jump} \; r, v; I) \;\longrightarrow\; (l'_{[0]}, H, R, I')}$$

[IF-FALSE]

$$\frac{R(r) \neq 0}{(l_{[n]}, H, R, \textbf{if-jump} \; r, v; I) \;\longrightarrow\; (l_{[n+1]}, H, R, I)}$$

[JUMP]

$$\frac{\hat{H}(R, v) = (l, [\star] \; I)}{(\bar{l}, H, R, \textbf{jump} \; v) \;\longrightarrow\; (l_{[0]}, H, R, I)}$$

[HALT]

$$(\bar{l}, H, R, \textbf{halt}) \;\longrightarrow\; (\bar{l}, H, R, \textbf{halt})$$

Figure 29. TPAL sequential transitions: $(\bar{l}, H, R, I) \;\longrightarrow\; (\bar{l}', H', R', I')$.

which to put an offset value. The offset value is set to be the address of the cell at which that promotion-ready mark that was popped.

D Example trace of `prod`

For our trace execution, we are going to assume that we are running our `prod` program in a TPAL task, whose register file R initially holds assignments for the argument values `a` $\mapsto$ 3 and `b` $\mapsto$ 4. In addition to the register file, there is the contents of the heartbeat cycle counter $\diamond$, a line number, corresponding to the program counter, and the instruction at that line number, which is being executed.

$\diamond$	0	1	2	3	4
R	`a` $\mapsto$ 3 `b` $\mapsto$ 4	`r` $\mapsto$ 0 ...	...	...	`r` $\mapsto$ 4 ...
line #	2	3	10	11	12
$\imath$	`r := 0`	**jump** `loop`	**if-jump** `a, exit`	`r := r + b`	`a := a - 1`

Note that, contrary to the C convention, our **if-jump** instruction branches to its argument label if its register argument is zero, and otherwise falls through.

The task parallelism in our `prod` program is based on the following observation: each iteration performed by the `loop` block can, in principle, be executed independently by parallel tasks. Of course, there is a potential obstacle: our `loop` block updates an accumulator register `r` in a serial fashion: $((\ldots(\texttt{0} + \texttt{b})\ldots) + \texttt{b}) + \texttt{b}$. But these updates can, thanks to the associativity of integer addition, break into smaller tasks, such as $(\texttt{0} + \ldots + \texttt{b}) + \ldots + (\texttt{0} + \ldots + \texttt{b})$. We just need a way of carving out pieces of work into fresh tasks and a way to combine the results of parallel tasks at join points. Our TPAL handles the creation of parallel tasks by heartbeat scheduling. In heartbeat scheduling, the program tries to create a new parallel task on a regular basis, at a rate that is controlled by the heartbeat interval, denoted $\heartsuit$. The heartbeat interval is a tuning parameter for heartbeat scheduling that is machine specific, and can be found by running a tuning program one time, on the target machine. Let us suppose that, for our sample execution, the setting $\heartsuit = 4$ is what we got from tuning, noting, however, that values in practice on real machines tend to be much larger.

[SEQ]

$$\frac{\neg\text{PromotionReady}(\bar{l}, H, \diamond) \quad (\bar{l}, H, R, I) \;\rightarrow\; (\bar{l}', H', R', I') \quad J; \langle \bar{l}'; \diamond + 1; H'; R'; I' \rangle \;\Downarrow\; J'; \langle \bar{l}''; \diamond'; H''; R''; I'' \rangle; g}{J; \langle \bar{l}; \diamond; H; R; I \rangle \;\Downarrow\; J'; \langle \bar{l}''; \diamond'; H''; R''; I'' \rangle; \mathbf{1} \cdot g}$$

[JRALLOC]

$$\frac{\neg\text{PromotionReady}(l_{[n]}, H, \diamond) \quad j \text{ fresh} \quad J' = \{\, j \mapsto (l_k; \textbf{jsclosed}) \,\} \cup J \quad R' = R[r = j] \quad J'; \langle l_{[n+1]}; \diamond + 1; H; R'; I \rangle \;\Downarrow\; J''; \langle \bar{l}'; \diamond'; H'; R''; I'' \rangle; g}{J; \langle l_{[n]}; \diamond; H; R; r \,:=\, \textbf{jralloc}\; l_k; I \rangle \;\Downarrow\; J'; \langle \bar{l}'; \diamond'; H'; R''; I'' \rangle; \mathbf{1} \cdot g}$$

[FORK]

$$\frac{\begin{array}{c} \neg\text{PromotionReady}(l_{[n]}, H, \diamond) \\ R(r) = j \quad J(j) = (l_k; js) \\ J_0 = \{\, j \mapsto (l_k; \textbf{jsopen}) \,\} \cup J \quad \hat{H}(R, v_t) = (l_t, [\star]\; I_t) \\ J_0; \langle l_{[0]}; 0; H; R; I \rangle \;\Downarrow\; J_1; \langle \bar{l}_1; \diamond_1; H_1; R_1; \textbf{join}\; r_1 \rangle; g_1 \\ J_0; \langle l_{t\,[0]}; 0; H; R; I_t \rangle \;\Downarrow\; J_2; \langle \bar{l}_2; \diamond_2; H_2; R_2; \textbf{join}\; r_2 \rangle; g_2 \\ R_1(r_1) = R_2(r_2) = j \\ H(l_k) = [\textbf{jtppt}\, jp; \Delta R; l_c]\; I' \quad H(l_c) = [\star_c]\; I_c \\ \text{MergeR}(R_1, R_2, \Delta R) = R_c \quad \text{MergeH}(H_1, H_2) = H' \\ \{\, j' \mapsto jr \mid (j' \mapsto jr) \in \text{MergeJ}(J_1, J_2) \wedge j' \neq j \,\} \cup \{\, j \mapsto (l_k; js) \,\} = J_c \\ J_c; \langle l_{c\,[0]}; 0; H'; R_c; I_c \rangle \;\Downarrow\; J'; T; g' \end{array}}{J; \langle l_{[n]}; \diamond; H; R; \textbf{fork}\; r, v_t; I \rangle \;\Downarrow\; J'; T; (g_1 \parallel g_2) \cdot g'}$$

[JOIN-BLOCK]

$$\frac{\neg\text{PromotionReady}(\bar{l}, H, \diamond) \quad R(r) = j \quad J(j) = (l_k; \textbf{jsopen})}{J; \langle \bar{l}; \diamond; H; R; \textbf{join}\; r \rangle \;\Downarrow\; J; \langle \bar{l}; \diamond; H; R; \textbf{join}\; r \rangle; \mathbf{1}}$$

[JOIN-CONTINUE]

$$\frac{\neg\text{PromotionReady}(\bar{l}, H, \diamond) \quad R(r) = j \quad J = (j \mapsto (l_k; \textbf{jsclosed})) \cup J' \quad H(l_k) = [\star]\; I \quad J'; \langle l_{k\,[0]}; \diamond; H; R; I \rangle \;\Downarrow\; J'; T'; g}{J; \langle \bar{l}; \diamond; H; R; \textbf{join}\; r \rangle \;\Downarrow\; J'; T'; \mathbf{1} \cdot g}$$

[TRY-PROMOTE]

$$\frac{\text{PromotionReady}(l_{[n]}, H, \diamond) \quad H(l) = [\textbf{prppt}\, l_h]\; I_h \quad J; \langle l_{h\,[0]}; 0; H; R; I_h \rangle \;\Downarrow\; J'; T; g}{J; \langle l_{[n]}; \diamond; H; R; I \rangle \;\Downarrow\; J'; T; \mathbf{1} \cdot g}$$

Figure 30. TPAL parallel transitions: $J; T \;\Downarrow\; J'; T'; g$.

D.1 Heartbeat interrupts

Continuing where we left off with the program trace, we see that the next instruction is going to jump back to the top of the `loop` block. However, before this jump happens, heartbeat scheduling takes over by interrupting our task, temporarily. This temporary interruption happens at this step because it is the first step for which the two following properties hold simultaneously. First, it is now the case that, based on our machine's tuning parameter $\heartsuit$, sufficient sequential work has been performed by our current task for it to amortize the creation of a new parallel task. In other words, the heartbeat cycle counter $\diamond = 5$ is greater than the heartbeat interval $\heartsuit = 4$. Second, the jump instruction brought the program counter to a

[STACK-NEW]
$$\frac{R[r = \textbf{uptr tup}\ ()] = R'}{(l_{[n']}, H, R, r\ :=\ \textbf{snew}; I)\ \longrightarrow\ (l_{[n'+1]}, H, R', I)}$$

[STACK-ALLOC]
$$\frac{\begin{array}{c} R(r) = \textbf{uptr tup}\ (v_0, \ldots, v_m) \\ R[r = \textbf{uptr tup}\ (\underbrace{0, \ldots, 0,}_{n} v_0, \ldots, v_m)] = R' \end{array}}{(l_{[n']}, H, R, \textbf{salloc}\ r, n; I)\ \longrightarrow\ (l_{[n'+1]}, H, R', I)}$$

[STACK-FREE]
$$\frac{\begin{array}{c} R(r) = \textbf{uptr tup}\ (v_0, \ldots, v_{n-1}, v_n, \ldots, v_m) \\ R[r = \textbf{uptr tup}\ (v_n, \ldots, v_m)] = R' \end{array}}{(l_{[n']}, H, R, \textbf{sfree}\ r, n; I)\ \longrightarrow\ (l_{[n'+1]}, H, R', I)}$$

[STACK-STORE]
$$\frac{\begin{array}{c} R(r) = \textbf{uptr tup}\ (v_0, \ldots, v_n, v_{n+1}, \ldots, v_m) \\ R[r = \textbf{uptr tup}\ (v_0, \ldots, v, v_{n+1}, \ldots, v_m)] = R' \end{array}}{(l_{[n']}, H, R, \textbf{mem}[r + n]\ :=\ v; I)\ \longrightarrow\ (l_{[n'+1]}, H, R', I)}$$

[STACK-LOAD]
$$\frac{R(r) = \textbf{uptr tup}\ (v_0, \ldots, v_n, v_{n+1}, \ldots, v_m) \quad R[r_d = v_n] = R'}{(l_{[n']}, H, R, r_d\ :=\ \textbf{mem}[r + n]; I)\ \longrightarrow\ (l_{[n'+1]}, H, R', I)}$$

[PRM-EMPTY-TRUE]
$$\frac{R(r) = \textbf{uptr tup}\ (v_0, \ldots, v_n) \quad \textbf{prmark} \in \{\, v_0, \ldots, v_n \,\} \quad R[r_d = 0] = R'}{(l_{[n']}, H, R, r_d := \textbf{prmempty}\ r; I)\ \longrightarrow\ (l_{[n'+1]}, H, R', I)}$$

[PRM-EMPTY-FALSE]
$$\frac{R(r) = \textbf{uptr tup}\ (v_0, \ldots, v_n) \quad \textbf{prmark} \notin \{\, v_0, \ldots, v_n \,\} \quad R[r_d = 1] = R'}{(l_{[n']}, H, R, r_d := \textbf{prmempty}\ r; I)\ \longrightarrow\ (l_{[n'+1]}, H, R', I)}$$

[PRM-PUSH]
$$\frac{\begin{array}{c} R(r) = \textbf{uptr tup}\ (v_0, \ldots, v_n, v_{n+1}, \ldots, v_m) \\ R[r = \textbf{uptr tup}\ (v_0, \ldots, \textbf{prmark}, v_{n+1}, \ldots, v_m)] = R' \end{array}}{(l_{[n']}, H, R, \textbf{prmpush}\ \textbf{mem}[r + n]; I)\ \longrightarrow\ (l_{[n'+1]}, H, R', I)}$$

[PRM-POP]
$$\frac{\begin{array}{c} R(r) = \textbf{uptr tup}\ (v_0, \ldots, v_n, v_{n+1}, \ldots, v_m) \quad \textbf{prmark} = v_n \\ R[r = \textbf{uptr tup}\ (v_0, \ldots, 0, v_{n+1}, \ldots, v_m)] = R' \end{array}}{(l_{[n']}, H, R, \textbf{prmpop}\ \textbf{mem}[r + n]; I)\ \longrightarrow\ (l_{[n'+1]}, H, R', I)}$$

[PRM-SPLIT]
$$\frac{\begin{array}{c} R(r_s) = \textbf{uptr tup}\ (v_0, \ldots, v_{n-1}, \textbf{prmark}, v_{n+1}, \ldots, v_m) \quad \textbf{prmark} \notin \{\, v_{n+1}, \ldots, v_m \,\} \\ R[r_s = \textbf{uptr tup}\ (v_0, \ldots, v_{n-1}, 0, v_{n+1}, \ldots, v_m)] = R' \quad R'[r_p = n] = R'' \end{array}}{(l_{[n']}, H, R, \textbf{prmsplit}\ r_s, r_p; I)\ \longrightarrow\ (l_{[n'+1]}, H, R'', I)}$$

Figure 31. TPAL sequential transitions for the stack: $(\bar{l}, H, R, I)\ \longrightarrow\ (\bar{l}', H', R', I')$.

promotion-ready program point. A promotion-ready program point is, in general, the first instruction in a block that has the **prppt** annotation. To see the heartbeat interrupt in action, let us continue with our execution trace where we left off, just after the jump instruction. At this point, our current task proceeds to handle the heartbeat interrupt by using the handler as specified by the annotation on the `loop` block. This handler block is the one labeled `loop-try-promote` in Figure 33. The first action carried out by the handler block is to test whether there is any latent parallelism held by the current task. This test is performed by first two instructions of the handler, which takes a branch based on the contents of `a`.

```
1  prod: [·] // computes c =  a * b
2    r := 0
3    jump loop
4
5  exit: [jtppt assoc-comm;
6              {r ↦ r2}; comb]
7    c := r
8    jump ret

9  loop: [prppt loop-try-promote]
10   if-jump a, exit
11   r := r + b
12   a := a - 1
13   jump loop
```

Figure 32. The sequential blocks of the prod program.

```
14 loop-try-promote: [·]
15   t := a < 2
16   if-jump t, loop
17   jr := jralloc exit
18   jump loop-promote
19
20 loop-par-try-promote: [·]
21   t := a < 2
22   if-jump t, loop-par
23   jump loop-promote

24 loop-promote: [·]
25   m := a / 2
26   n := a % 2
27   a := m
28   tr := r
29   r := 0
30   fork jr, loop-par
31   a := m + n
32   r := tr
33   jump loop-par
```

Figure 33. Promotion-handler blocks of prod.

5	6	7
a ↦ 2		t ↦ 1
...	...	...
13	15	16
jump loop	t := a < 2	**if-jump** t, loop

At this point, if the number of iterations in a were fewer than two, then, in effect, the conditional branch would transfer control away from the handler, and back to the loop, by branching to the loop block. However, given that there remain a = 2 iterations in the trace, the handler can commit to releasing some of the parallelism that is latent in the loop.

D.2 Promotion

In TPAL, the process of taking latent parallelism, e.g., the ≥ 2 iterations remaining in register a, and transfering some to a fresh parallel task is called *promotion*. The first promotion performed by a parallel region is special, because the task initiating the process has to create a new synchronization variable to coordinate between all the parallel tasks that might be involved. The next instruction allocates a new *join record*, which is a synchronization variable that is going to be used by the parent and its child tasks. The exit block label that is being passed to **jralloc** specifies a custom protocol that will later manage the aggregation of results and the continuation of the parallel tasks involved in the join.

8	9
	jr ↦ j0
...	...
17	18
jr := **jralloc** exit	**jump** loop-promote

In the next step, there appears in the register file a pointer to the join record, j0, which was allocated in the heap. With the join pointer ready, the handler jumps to the loop-promote block, where the promotion will be completed. The loop-promote block first performs a sequence of instructions, from line 25 to 29, to prepare the registers of the current task to seed a new child task. After the initial instructions, the child task is spawned as a side effect of executing the **fork** instruction. This instruction takes two arguments, a pointer to a join record, jr, and the label of a block, loop-par. Its first action is to register a dependency edge in the join record for the child task. Its second action is to spawn the child task, with a copy of the register file of the parent task, but with its program counter assigned to execute the block at the argument label, loop-par. Finally, the fork instruction continues executing the parent task from the next instruction, starting with a fresh heartbeat cycle counter, i.e., ⋄ = 0.

```
34 loop-par: [prppt loop-par-try-promote]
35   if-jump a, exit-par
36   r := r + b
37   a := a - 1
38   jump loop-par

39 comb : [·]
40   r := r + r2
41   join jr
42
43 exit-par: [·]
44   join jr
```

Figure 34. Parallel blocks of prod.

15	0	1	2
m ↦ 1 n ↦ 0 a ↦ 1 tr ↦ 4 r ↦ 0 ...	...	a ↦ 1 ...	r ↦ 4 ...
30	31	32	33
fork jr, loop-par	a := m + n	r := tr	**jump** loop-par

After completing fork instruction, the parent task issues the final instructions of the heartbeat interrupt handler. These instructions rearrange the registers so that the parent task can resume executing its part of the computation, with its original accumulator value back in r.

D.3 Parallel tasks

The child and parent tasks are now executing in parallel, but have resumed executing from a different loop block, namely loop-par, which resides in a different parallel region of the program. The parallel region of our prod program consists of the three blocks shown in Figure 34. Let us continue our trace, this time from the starting point of the child task that our parent task just spawned.

0	1	2	3
a ↦ 1 ...	...	r ↦ 4 ...	a ↦ 0 ...
35	36	37	38
if-jump a, exit-par	r := r + b	a := a - 1	**jump** loop-par

The sequence of instructions executed by the child task looks almost identical to the ones in the loop block, where our trace started, i.e., in the parent task, before the heartbeat interrupt arrived. The difference is in the continuation of the loop: before it can exit with a result, the prod program has to aggregate the results of all the parallel tasks. In our trace, the only other task involved in the join is the parent task, which has since been executing in parallel from where we left off. Thus far, the parent thread has executed with almost the same trace as our child task, except starting with a different heartbeat cycle counter. There is a slightly larger value accumuated in the heartbeat cycle counter ⋄ of the parent task due to there being extra instructions at the end of the heartbeat handler, just before jumping to loop-par. As a consequence, the parent task triggers one additional heartbeat interrupt, whereas the child completes before triggering one. The interrupt handled by the parent task does not lead to a promotion in our trace, because the value of a in the register file of the parent task is zero by this point. Consequently, the parent task executes the handler up to line 22, where the handler short circuits.

In general, however, if either child or parent tasks were holding onto at least two remaining iterations, our program may generate a dynamic number of additional tasks. Any such additional tasks are going to be spawned with the same join record, that is, the one allocated by the initial invocation of the heartbeat handler, loop-try-promote. To get this behavior, we structured our prod program so that all subsequent heartbeat interrupts trigger a different handler block, namely loop-par-try-promote, shown in Figure 33. This handler assumes that the join record is already referenced by register jr. Such handling of join records is essential: in TPAL every dynamic instance of a parallel region, such as the one we traced thus far, acts on one join record. In general, however, there may be multiple dynamic instances of the same parallel region, e.g., in a program that executes multiple instances of our prod program. In such a case, there can be a dynamic number of join records in flight at a time, one for each dynamic instance of prod, for example.

D.4 Join resolution

Our sample program is nearly finished, except for the final steps, whereby all of the parallel tasks take part in aggregating their partial results, after which a final result is obtained. Let us suppose that the child task is going to be the first to reach its join point. Resuming where we left off above, our child task branches out of its loop to the `exit-par` block, where it executes the **join** instruction.

4	5
...	...
35	44
if-jump `a, exit-par`	**join** `jr`

At this point, the join between the child and parent tasks is *open*, because there is at least one task that has not yet reached the join point. Because the join is open, the child task is removed from the task graph, and its register file is stashed away, e.g., in the join record, for later. Later, after the parent task reaches its joint point, the behavior changes, because the parent can now combine its result, in `r`, with the result computed by the child task. The next steps of the parent task are all specific to the *join-resolution protocol* that is determined by the label that was originally passed to **jralloc**, in our case `exit`. The join-resolution protocol is customized by the annotation on this `exit` block. The annotation determines, first, how to combine the results of the tasks by specifying a code block, which is in our case the `comb` block. There is, however, one complication: because our `comb` block needs the values from the `r` registers of both tasks, there needs to be one extra register to pass the result from the child task. The annotation determines, second, what register will be seeded with the child copy of `r`, which is in our case `r2`. After aggregating the results, the `comb` block itself issues the **fork** instruction one last time, after which, the join becomes *closed*. Because the join point is closed, the parent task takes ownership of the continuation by jumping to the `exit` block.

0	1	2	3	4
		$\mathtt{r} \mapsto 8$ $\mathtt{r2} \mapsto 4$	$\mathtt{r} \mapsto 12$	
...	...	...	...	...
35	44	40	41	7
if-jump `a, exit-par`	**join** `jr`	`r := r + r2`	**join** `jr`	`c := r`

Even though it is the parent task in this case, any one of the tasks involved in a join can, depending on which finishes last, be the one to close the join and execute the continuation. Our parallel program has now reached its end: it will store the final result in `c` and jump to the continuation block, `ret`.

D.5 Discussion

The diagram in Figure 35 shows the control-flow graph of the entire `prod` program. Most edges in the graph represent control flow that is triggered by conditional (e.g., `loop` → `loop`) and unconditional jump instructions (e.g., `prod` → `loop`). The other edges are the result of a heartbeat interrupt (e.g., `loop` → `loop-try-promote`), a fork instruction (e.g., `loop-try-promote` → `loop-par`), or a join instruction (e.g., `comb` → `exit-par`). With this representation in mind, we can now address one obvious question: Why are `loop` and `loop-par` so similar? Could we do with just one? Indeed, we could have written the program with just one loop block, but such a representation brings with it some indirect costs. If we insist on there being one loop block, then we either must allocate the join record `jr` up front, before the loop starts, or delay allocating the join record by initially storing a sentinel value in `jr`, and updating `loop-try-promote` to handle both sentinel and non-sentinel cases. Allocating the join record up front is undesirable because, even if its cost is small, small costs can easily add up to a large overall cost. Delaying the allocation helps avoid the cost of allocation, but still requires there to be, e.g., a conditional branch on the exit of the loop, because the loop needs to issue a join instruction if and only if at least one promotion occurred.

In general, the two practical options are the *expanded style* we used for our `prod` program, or the *reduced style* that would take the approach of delaying the allocation of the join record. There is a tradeoff between the two options that we need to consider. In the expanded style, we can see that there is at least one opportunity for code specialization. That is, in our program `prod` program, the blocks `prod`, `loop` and `exit` pay zero overhead cost for task parallelism (the **prppt** and **jtppt** annotations carry zero runtime overhead). In particular, these blocks do not see the register `jr` as alive, and therefore never have to pay to access it. This property is crucial for being able to scale down: for instances of `prod` that are sufficiently small (e.g., small enough to complete in between successive heartbeat interrupts), the three sequential blocks may complete their computation without being involved in a promotion. Overall, the expanded style can achieve the cleanest separation between parallel and sequential code, ultimately providing a complete statically identifiable part of the control-flow graph, for which we can avoid overheads, such as the management of the join counter, and more generally use of deeper specializations. Deeper

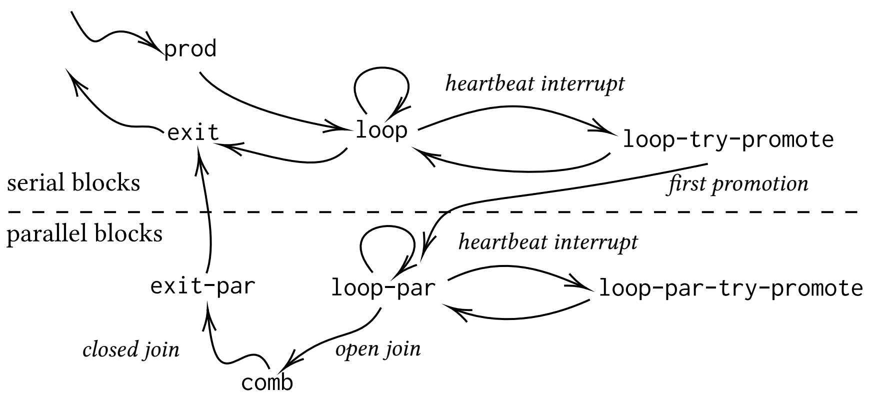


Figure 35. Control-flow graph of our prod program.

specializations might, for example, open up opportunities to optimize the sequential part of the control-flow graph more aggressively than otherwise, because in such blocks, it is guaranteed that the loop up to that point has run in a completely serial fashion since it began. Specializations such as these do cause an increase in code size, an issue that has implications for nested parallel loops, which we discuss next.