1

ENABLING PRIMITIVES FOR COMPILING PARALLEL LANGUAGES

Seth Copen Goldstein, Klaus Erik Schauser*, David Culler

Computer Science Division, University of California at Berkeley
{sethg,culler}@cs.berkeley.edu
**Department of Computer Science, University of California at Santa Barbara*
schauser@cs.ucsb.edu

ABSTRACT

This paper presents three novel language implementation primitives—lazy threads, stacklets, and synchronizers—and shows how they combine to provide a parallel call at nearly the efficiency of a sequential call. The central idea is to transform parallel calls into parallel-ready sequential calls. Excess parallelism degrades into sequential calls with the attendant efficient stack management and direct transfer of control and data, unless a call truly needs to execute in parallel, in which case it gets its own thread of control. We show how these techniques can be applied to distribute work efficiently on multiprocessors.

1 INTRODUCTION

Many modern parallel languages provide methods for dynamically creating multiple independent threads of control, such as forks, parallel calls, futures [15], object methods, and non-strict evaluation of argument expressions [17, 12]. Generally, these threads describe the logical parallelism in the program. The programming language implementation maps this dynamic collection of threads onto the fixed set of physical processors executing the program, either by providing its own language-specific scheduling mechanisms or by using a general threads package. These languages stand in contrast to languages with a single logical thread of control, such as Fortran90, or a fixed set of threads, such as Split-C. There are many reasons to have the logical parallelism of the program exceed the physical parallelism of the machine, including ease of expressing parallelism and better utilization in the presence of synchronization delays [16, 25], load imbalance, and long communication latency. Moreover, the semantics of the language or the synchronization primitives may allow dependencies to be expressed in such a way that progress can be made only by interleaving multiple threads, effectively running them in parallel even on a single processor.

A parallel call is fundamentally more expensive than a sequential call because of the storage management, data transfer, scheduling, and synchronization involved. This cost has been reduced with a combination of compiler techniques and clever run-time representations [7, 19, 23, 16, 25, 20, 18], and by supporting fine-grained parallel execution directly in hardware [13, 2]. These approaches, among others, have been used in implementing the parallel programming languages Mul-T [15], Id90 [7, 19], CC++ [5], Charm [14], Cilk [3], Cid [18], and Olden [4]. In many cases, the cost of the parallel call is reduced by severely restricting what can be done in a thread.

In earlier approaches, the full cost of parallelism is borne for all potentially parallel calls, although the parallelism is neither needed nor exploited in most instances. For example, once all the processors are busy, there may be no need to spawn additional work, and in the vast majority of cases the logic of the program permits the child to run to completion while the parent is suspended. The goal of this work is to make the cost of a potentially parallel call as close as possible to that of a sequential call unless multiple threads of control or remote execution are actually needed. We also produce a very fast parallel call when it is needed.

The key idea is that we fork a new thread as if it were a sequential call and elevate it to a true fork of a local thread only if the child actually suspends. This concept, which we call *lazy threads*, builds upon work on lazy task creation [15]. In the best case our system eliminates all the run-time bookkeeping costs associated with forking a thread or creating a future. In the worst case it requires only three instructions to create a future. Similarly, we can defer generating work for other processors until a request for work is received from another processor. If all the processors have plenty to do, potential parallel work is simply assumed by the current thread of control. Our experience is that potentially parallel calls frequently degenerate into the simple, local, sequential case and that handling the simple case very well has a significant impact on performance.

Our current experimental results focus on two prototype implementations on the CM-5: a direct implementation in C and a compiler for the fine-grained parallel language Id90. The C implementation was used to write some kernels and shows that these primitives introduce little or no overhead over sequential programs. The Id90 implementation shows that for complete programs we achieve a substantial improvement over previous work. Our work is applicable to many other systems as well. For example, our techniques could be applied to other programming languages [5, 26], thread packages [8], and multithreaded execution models. Our work relies extensively on compiler optimizations; lazy threads cannot simply be implemented with a function call in a user-level threads library without substantial loss of efficiency. Because the synthesis between compiler and run-time system is key to obtaining efficiency, these ideas must be evaluated in the context of an actual compiler.

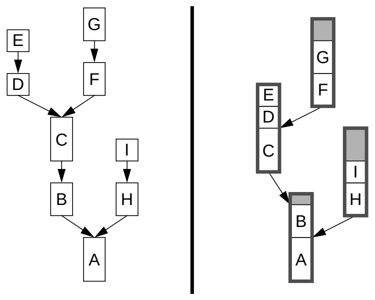


Figure 1 How individual activation frames of a cactus stack are mapped onto stacklets. Only parallel calls or stacklet overflows require allocation of a new stacklet. (The arrows point back to the parent; in the above example, A calls B and H in parallel.)

1.1 Overview

In this work a *thread* is a locus of control on a processor which can perform calls to arbitrary nesting depth, suspend at any point, and fork additional threads. Threads are scheduled independently and are non-preemptive.[1] We associate with each thread its own unbounded stack.

Before considering the parallel call, observe that the efficiency of a sequential call derives from several cooperating factors. Storage allocation on call and return involves merely adjusting the stack pointer, because the parent is suspended upon call and the child and its children have completed upon return. Data and control are transferred together on the call and on the return, so arguments and return values can be passed in registers and no explicit synchronization is involved.

To realize a parallel-ready sequential call—i.e., one that creates a sequential task that can be elevated gracefully into an independent thread of control—we proceed in four steps.

First, in Section 2, we address storage allocation. Since threads can fork other threads and each requires a stack, a tree of stacks, a *cactus stack*, is required. We realize this cactus stack using *stacklets* (see Figure 1). A stacklet is a fixed-size region of storage which can contain multiple call frames, is cheap to allocate, and is managed internally like a sequential stack. Allocation of a new stacklet occurs when a new thread is created or when a stacklet overflows. *Stacklet stubs* are used to handle many special cases, including underflow and remote return, without the sequential call needing to perform tests on return. This provides a naive parallel language implementation with conventional local and remote forks.

Next, in Section 3, we address control and data transfer when a thread is forked on the local processor. A *lazy thread fork* is performed exactly like a sequential call; control

[1] This is similar to what is provided in many kernel threads packages. Our threads, however, are stronger than those in TAM [7] and in some user-level threads packages, e.g. Chorus [21], which require that the maximum stack size be specified upon thread creation so that memory can be preallocated.

and data are transferred to the child and the call is made on the parent stack. However, if the child suspends, the parent is resumed with its stack extended, so it gives up its own stacklet to the new thread and uses a new stacklet for its subsequent children. We generate code so that the child can suspend and resume the parent by doing a jump to a simple offset of the return address. Thus, control is transferred directly to the parent on either return or suspension.

Third, in Section 4, we show how to perform synchronization cheaply between the parent and child, should they become independent threads. The only flexibility we have in the sequential call is the indirect jump on the return address. The key idea, implemented by *synchronizers*, is that the parent and the child share the return address, which by our code generation technique represents multiple return addresses. The return entry points can be adjusted to reflect the synchronization state. The optimizations outlined so far are required to support many logical threads on a single processor.

Finally, in Section 5, we extend the use of multiple return addresses to allow the parent to generate additional parallel work *on demand*, in response to a work-stealing request from another processor. We call this concept a thread *seed* because it allows potential threads to be held dormant very cheaply until they are either assumed by the local processor or stolen and planted in another processor. Growing a thread seed into a full thread requires executing a piece of code in the context of the function that created it. On the other hand, the overhead for creating and assuming a thread seed is minimal.

In Section 6 we give empirical data to show that these concepts can be combined to efficiently implement excess logical parallelism. Underlying our optimizations is the observation that in modern microprocessors, a substantial cost is paid for memory references and branches, whereas register operations are essentially free. Since stacklets are managed like a sequential stack, arguments and results can be passed in registers, even in the potentially parallel case. By manipulating the existing indirect return jump, conditional tests for synchronization and special cases can be avoided.

1.2 Related Work

Attempts to accommodate logical parallelism have include thread packages [8, 21, 6], compiler techniques and clever run-time representations [7, 19, 16, 25, 23, 20, 10], and direct hardware support for fine-grained parallel execution [13, 2]. These approaches have been used to implement many parallel languages, e.g. Mul-T [15], Id90 [7, 19], CC++ [5], Charm [14], Cilk [3], Olden [4], and Cid [18]. The common goal is to reduce the overhead associated with managing the logical parallelism. While much of this work overlaps ours, none has combined the techniques described in this paper into an integrated whole. More importantly, none has started from the premise that all calls, parallel or sequential, can be initiated in the exact same manner.

Our work grew out of previous efforts to implement the non-strict functional language Id90 for commodity parallel machines. Our earlier work developed a *Threaded Abstract Machine* (TAM) which serves as an intermediate compilation target [7]. The two key differences between this work and TAM are that under TAM calls are always parallel, and due to TAM's scheduling hierarchy, calling another function does not immediately transfer control.

Our lazy thread fork allows all calls to begin in the same way, and creates only the required amount of concurrency. In the framework of previous work it allows excess parallelism to degrade efficiently into a sequential call. Many other researchers have proposed schemes which deal lazily with excess parallelism. Our approach builds on *lazy task creation* (LTC) which maintains a data structure to record previously encountered parallel calls [16]. When a processor runs out of work, dynamic load balancing can be effected by stealing previously created lazy tasks from other processors. These ideas were studied for Mul-T running on shared-memory machines. The primary difference is that LTC always performs extra work for parallel calls, whether they execute locally or remotely. Even the lazy tasks that are never raised to full fledged tasks are "spawned off" in the sense that they require extra bookkeeping. In addition, in order to avoid memory references and increase efficiency our work uses different primitives from LTC. LTC also depends on a garbage collector, which hides many of the costs of stack management. Finally, while earlier systems based on LTC relied on shared-memory hardware capabilities, our implementation works on both distributed- and shared-memory systems.

Another proposed technique for improving LTC is *leapfrogging* [25]. Unlike the techniques we use, it restricts the behavior of the program in an attempt to reduce the cost of futures.

We use stacklets for efficient stack-based frame allocation in parallel programs. Previous work in [10] developed similar ideas for handling continuations efficiently. Olden [4] uses a "spaghetti stack." In both systems, the allocation of a new stack frame always requires memory references and a garbage collector.

The way thread seeds encode future work builds on the use of multiple offsets from a single return address to handle special cases. This technique was used in SOAR [22]. It was also applied to Self, which uses parent controlled return continuations to handle debugging [11]. We extend these two ideas to form synchronizers.

Building on LTC, Olden [20, 4] applies similar techniques for the automatic parallelization of programs using dynamic data structures. Of the systems mentioned so far, Olden's integration is closest to ours.

Finally, user-level thread packages are still not as lightweight as many of the systems mentioned above. Since the primitives of thread packages are exposed at the library level, the compiler optimizations we present are not possible for such systems.

2 STORAGE MANAGEMENT: STACKLETS

Stacklets are a memory management primitive which efficiently supports cactus stacks. Each stacklet can be managed like a sequential stack. A stacklet is a region of contiguous memory on a single processor that can store several activation frames (see

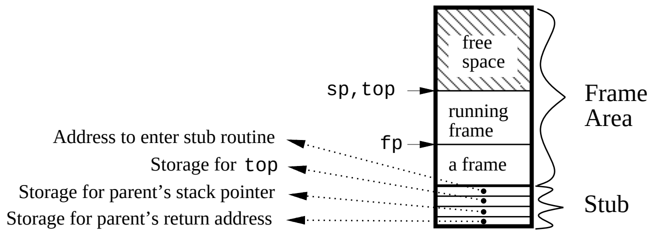

free space
sp,top
running frame
fp
a frame
Frame Area
Stub
Address to enter stub routine
Storage for top
Storage for parent's stack pointer
Storage for parent's return address

Figure 2 The basic form of a stacklet.

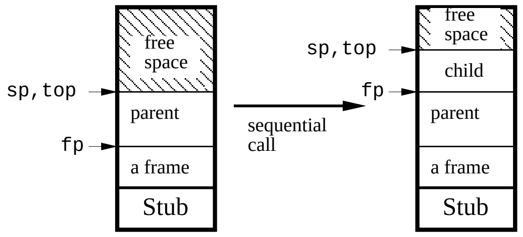


Figure 3 The result of a sequential call which does not overflow the stacklet.

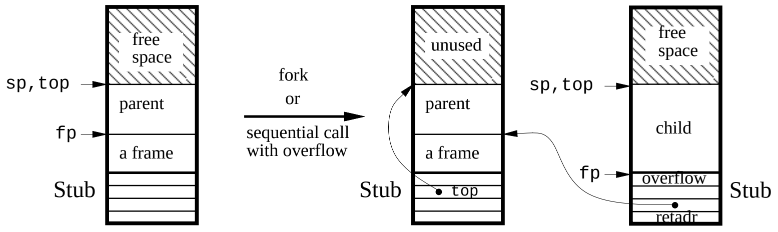


Figure 4 The result of a fork or of a sequential call which overflows the stacklet.

Figure 2). Each stacklet is divided into two regions, the stub and the frame area. The stub contains data that maintains the global cactus stack by linking the individual stacklets to each other. The frame area contains the activation frames. In addition to a traditional stack pointer (`sp`) and frame pointer (`fp`), our model defines a *top pointer* (`top`) which—for reasons presented in the next section—points to the top of the currently used portion of the stacklet, or, in other words, to the next free location in the stacklet. These three pointers are kept in registers.

We recognize three kinds of calls—sequential call, fork, and remote fork—each of which maps onto a different kind of allocation request. A sequential allocation is one that requests space on the same stack as the caller. The child performs the allocation; therefore, it determines whether its frame can fit on the same stacklet. If so, `sp`, `fp`, and `top` are updated appropriately (see Figure 3). If not, a new stacklet is allocated and the child frame is allocated on the new stacklet (see Figure 4). This also happens for a fork, which causes a new stacklet to be created on the local processor. We could

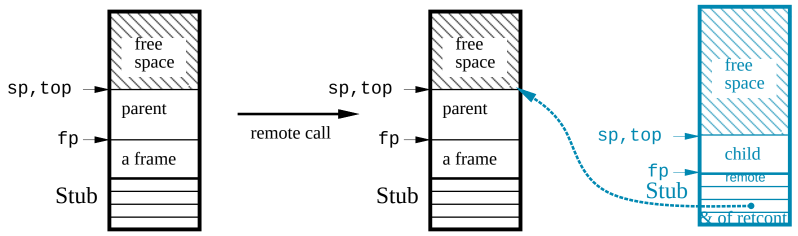


Figure 5 A remote fork leaves the current stacklet unchanged and allocates a new stacklet on another processor.

either run the child in the new stacklet immediately or schedule the child for later execution. In the former case, `fp`, `sp`, and `top` would point to the child stacklet (see Figure 4). In the latter case, they would remain unchanged after the allocation. For a remote fork there are no stacklet operations on the local processor. Instead, a message is sent to a remote processor with the child routine's address and arguments (see Figure 5).

In our current naive implementation, the overhead in checking for stacklet overflow in a sequential call is two register-based instructions (an AND of the new `sp` and a compare to the old) and a branch (which will usually be successfully predicted). If the stacklet overflows, a new stacklet is allocated from the heap. This cost is amortized over the many invocations that will run in the stacklet.

2.1 Stacklet Stubs

Stub handlers allow us to use the sequential return mechanism even though we are operating on a cactus stack. The stacklet stub stores all the data needed for the bottom frame to return to its parent. When a new stacklet is allocated, the parent's return address and frame pointer are saved in the stub and a return address to the stub handler is given to the child. When the bottom frame in a stacklet executes a return, it does not return to its caller; instead it returns to the stub handler. The stub handler performs stacklet deallocation and, using the data in the stacklet stub, carries out the necessary actions to return control to the parent (restoring `top`, and having `sp` and `fp` point to the parent).

In the case of a remote fork, the stub handler uses indirect active messages [24] to return data and control to the parent's message handler, which in turn has responsibility for integrating the data into the parent frame and indicating to the parent that its child has returned.

2.2 Compilation

To reduce the cost of frame allocation even further we construct a call graph which enables us to determine for all but the recursive calls whether an overflow check is needed. Each function has two entry points, one that checks stacklet overflow and another that does not. If the compiler can determine that no check is needed, it uses the latter entry point. This analysis inserts preventive stacklet allocation to guarantee that future children will not need to perform any overflow checks.

2.3 Discussion

In summary, stacklets provide efficient storage management for parallel execution. In the next section we will see that potentially parallel calls can use the same efficient mechanism as regular sequential calls, because each stacklet preserves the invariants of a stack. Specifically, the same call and return mechanisms are used; arguments and results can be passed in registers. These benefits are obtained at a small increase to the cost of sequential calls, namely checking whether a new stacklet needs to be allocated in the case of an overflow or parallel call. The extra cost amounts to a test and branch along with the use of an additional register. This overhead is required only when the compiler cannot statically determine that no check is needed. Stubs eliminate the need to check for underflows. This contrasts with previous approaches which always require some memory touch operations or a garbage collector.

3 CONTROL TRANSFER: THE LAZY THREAD CALL

Our goal is to make a fork as fast as a sequential call when the forked child executes sequentially. Using stacklets as the underlying frame-storage allocation mechanism gives us a choice as to where to run the new thread invoked by the fork. The obvious approach is to explicitly fork the new thread using the parallel allocation explained in the previous section. However, if the child is expected to complete without suspending—i.e., if it behaves like a sequential call—we would rather treat it like a sequential call and invoke the child on the current stacklet.

This section introduces a *lazy thread fork* (`tfork`) which behaves like a sequential call unless it suspends, in which case—in order to support the logical parallelism implied by the fork it represents—it directly resumes the parent and behaves like an eagerly forked thread. `tfork` behaves like a sequential call in that it transfers control (and its arguments) directly to the new thread. Further, if the new thread completes without suspending, it returns control (and results) directly to its parent.

If the child suspends, it must resume its parent in order to notify its parent that the `tfork` really required its own thread of control. Thus, the child must be able to return to its parent at either of two different addresses: one for normal return and one for suspension. Instead of passing the child two return addresses, the parent calls the child with a single address from which it can derive both addresses. At the implementation level, this use of multiple return addresses can be thought of as an extended version of continuation passing [1], where the child is passed two different continuations, one for normal return and one for suspension. The compiler ensures that the suspension entry point precedes the normal return entry point by a fixed number of instructions.

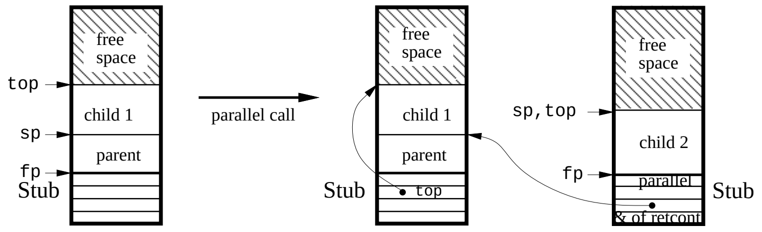


Figure 6 A parallel call creates a new stacklet.

A normal return will continue execution after the `tfork`. In the case of suspension, the compiler uses simple address arithmetic to calculate the suspension entry point.

In the case where the child suspends, the parent will not be the topmost frame in the stacklet—i.e., `sp` will not equal `top` (the situation shown in Figure 6). To maintain the sequential stack invariant, we do not allocate future children of the parent on the current stacklet. Instead, while there is a suspended child above the parent in the current stacklet, we allocate future children, sequential or parallel, on their own stacklets (see Figure 6). As a result, the translation for a call must first compare `sp` and `top`. If they are equal, the call occurs on the current stacklet (as in Figure 3). If they are different, it starts a new stacklet (as in Figure 6). As a result, regardless of the children's return order, no stacklet will ever contain free space between allocated frames. This simplifies memory management.

In summary, `tfork` allows a potentially parallel thread to be executed sequentially and still have the ability to suspend and get its own thread of control. A child that needs its own thread of control takes over its parent thread, causing its parent to allocate subsequent work on other stacklets. Using stacklets and compiler support we have created a multithreaded environment in a single address space which gives each thread a logically unbounded stack.

3.1 Parent Controlled Return Continuations

To reduce the cost of parallel calls, we always want a child which terminates to return directly to its parent. If the child terminates without suspension it can use its original return address. But if it suspends, is later resumed, and finally terminates, it generally cannot return to the same point: Once the child has suspended, the child and parent are truly separate threads and the parent may have already carried out the work immediately following the `tfork` that created the child.

If we want the child to return directly to the parent we need some way to modify the child's return continuation so that the child will return to a location of the parent's choosing. Since we are using stacklets, both the parent and the child know where the child's return address is located in the stack. If the parent is given permission to change the return continuation, it can change it to reflect the new state of the parent's

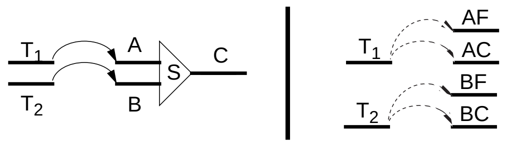


Figure 7 Example of a two-way join, illustrating synchronizers.

computation. The new return continuation will return the child to the point in the parent function that reflects that the child, and any work initiated after the child suspended, have been carried out. We call this mechanism *parent controlled return continuations* (PCRCs).

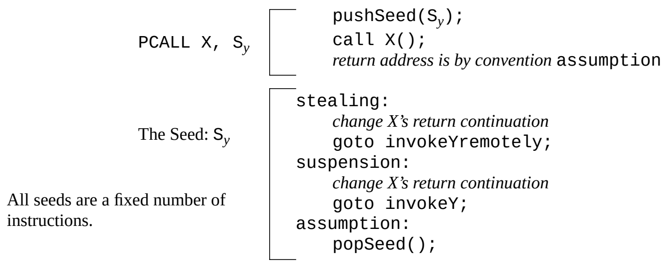
4 EFFICIENT SYNCHRONIZATION: SYNCHRONIZERS

The ideas found in PCRCs can be extended to efficiently implement synchronization. We minimize the synchronization cost due to joins by extending the use of PCRCs with judicious code duplication and by exploiting the flexibility of the indirect jump in the return instruction. The basic idea is to change the return continuation of a child to reflect the synchronization state of the parent. In this way, neither extra synchronization variables nor tests are needed. The amount of code duplicated is small since we need to copy only the code fragments that deal with returned results. This allows us to combine the return with synchronization at no additional run-time cost. For full details see [9].

Figure 7 illustrates synchronizers. As indicated in the left part of the figure, assume that we have two threads, T_1 and T_2, which return to the code fragments A and B, respectively, in the parent. There they synchronize, before starting the code C.[2] The key observation is that modifying the contents of the return address lets us encode synchronization information. PCRCs allow the parent to perform this modification.

The resulting situation, which relies on code duplication, is shown in the right part of Figure 7. Depending on the synchronization state, each of the two threads returns directly to the code for synchronization failure (AF or BF) or success (AC or BC). If both threads are explicitly forked, the return addresses for both initially point to their respective failure entry points (AF and BF). Whichever returns first executes its return code fragment (A or B) followed by a piece of code for synchronization failure. If the first thread was invoked with `tfork`, its initial failure entry point will invoke the second thread. The failure entry point will also modify the other thread's return address to point to the code for synchronization success. Synchronizers and PCRCs provide the mechanisms to efficiently combine the return and synchronization.

[2] Each of the three pieces of code is fairly short. A and B usually handle only taking the results and depositing them into the parent's frame, while C includes only one basic block.



```
PCALL X, S_y        pushSeed(S_y);
                    call X();
                    return address is by convention assumption

The Seed: S_y     stealing:
                    change X's return continuation
                    goto invokeYremotely;
                  suspension:
                    change X's return continuation
                    goto invokeY;
                  assumption:
                    popSeed();
```

Figure 8 How `pcall` and thread seeds are implemented.

5 REMOTE WORK: THE LAZY PARALLEL CALL

So far we have shown how to reduce the overhead of potential parallelism when it actually unfolds sequentially on the local processor. Here we extend these ideas to generate work for remote processors. We introduce *thread seeds*, which allow us to represent the potential work in the system so that it may be distributed efficiently among multiple processors. Our goal is to allow work to be distributed remotely, but pay the cost only when there is an actual need to do so.

Thread seeds are a direct extension of the multiple return addresses we introduced to handle the suspension of children invoked by a `tfork`. As shown in Figure 8, a thread seed is a code fragment with three entry points: one for child return, one for child suspension, and one for an external work-stealing request. At the implementation level, a thread seed can be thought of as an extended version of continuation passing [1], where the child is passed three different continuations, one for each of the three cases. When the compiler determines that there is work that could be run in parallel, it creates a thread seed which represents the work. For example, with two successive forks, the `tfork` for the first thread will be associated with a thread seed representing the fork of the second thread.

We combine seed generation and `tfork` into a single primitive, `pcall X,`s_Y, where `X` is the function to call and s_Y is a thread seed that will, when executed in the context of the parent, cause the function `Y` to be invoked. Upon execution of the `pcall`, a seed is created and control is transferred to `X`, making the current (parent) frame inactive. The newly created seed remains dormant until one of three things happens: the child returns (the seed is inlined), the child suspends (the seed is activated), or a remote processor requests work (the seed is stolen). All three cases require the intervention of the parent. If the child returns, the parent picks up the seed and inlines the new thread of control into its own, i.e.`Y` executes on the parent's stacklet, just as if it had been called sequentially. If the child suspends, the parent activates the seed by spawning `Y` off on its own new stacklet; the seed becomes a new thread concurrent with the first, but on the same processor. If a remote processor requests work, the parent executes a remote call of `Y`, which becomes a new thread running concurrently with the current thread, but on another processor.

The model described above is a direct extension of the multiple return addresses used to implement the lazy thread fork. In addition to two continuations for handling the "return" and "suspension," we need a third for "finding work." If a remote work request is received, the run-time system must somehow find the thread seed (the third continuation) to initiate the creation of remote work. Here we consider two approaches to finding such work: an implicit seed model and an explicit seed model.

In the implicit model, the remote work request interrupts the child, which then continues execution at the work-stealing entry point of its parent. If there is no work, the entry point contains a code fragment to jump to the parents ancestor on the stack. The search continues until either work is generated or no work is found because no excess parallelism is present. For the implicit model, the `pushSeed` and `popSeed` macros in Figure 8 turn into nops and the planting of a seed is an abstract operation. The advantage of this model is that when a `pcall` is made no bookkeeping is required. Instead, the stack frames themselves form the data structure needed to find work. The disadvantage is that finding work is more complex.

In the explicit model, when a seed is planted a special continuation is pushed onto the top of a *seed queue*. The continuation is a pointer to the return address in the frame. The calling convention is such that the return from the child will default to the assumption point. If a child suspends, it saves the top pointer in the stacklet stub, pops the top seed off the queue, sets the `sp` as indicated by the seed, and jumps into the suspension entry point of the seed.

The explicit queueing of seeds allows us to find work with just a few instructions. For instance, if a child suspends, it can find its ancestor, which has more work to perform, merely by popping off the top seed. Or, more importantly, if a remote processor requests work, we can determine if there is work by simply comparing the top and bottom pointers to the seed queue. We can also spawn off that work by jumping through the work-stealing entry point of the seed at the bottom of the queue. The parent, invoked through the seed, will execute the work-stealing routine, placing any appropriate seed on the bottom of the queue. The drawback of this scheme is that even when a seed is inlined into the current thread (the sequential case) there is an extra cost of two memory references over the previously described implicit scheme.

6 EXPERIMENTAL RESULTS

In this section we present preliminary performance results for our techniques on both uni- and multiprocessors. Our uniprocessor data were collected on a SparcStation 10. Our multiprocessor data were collected on a CM-5.

We have produced a parallel version of C for the CM-5 which incorporates the techniques presented in this paper. To evaluate these techniques we begin by comparing the performance of four different implementations of the doubly recursive Fibonacci function. Fib, being fine-grained, is a good "stress test" of function invocation. As shown in Table 1, the C version is significantly slower than either the synchronizer or

Compilation Method	Runtime (secs)
gcc -O4 fib.c	2.29
Assembly version of Fib	1.22
Fib with stacklets, lazy threads, and synchronizers	1.50
Fib as above with explicit seeds	1.86

Table 1 Comparing runtimes of fib 31 on a SparcStation 10.

Program	Short Description	Input Size	TAM	Lazy Threads
Gamteb	Monte Carlo neutron transport	40,000	220.8	139.0
Paraffins	Enumerate isomers of paraffins	19	6.6	2.4
Simple	Hydrodynamics and heat conduction	1 1 100	5.0	3.3
MMT	Matrix multiply test	500	70.5	66.5

Table 2 Dynamic run-time in seconds on a SparcStation 10 for the Id90 benchmark programs under the TAM model and lazy threads with multiple strands using explicit seeds. The programs are described in [7].

the seed version. The reason is that our stacklet management code does not use register windows, which introduce a high overhead on the Sparc. For a fair comparison we wrote an assembly version of Fib that also does not use register windows. This highly optimized assembly version runs only 18% faster than the synchronizer version, which incorporates all the mechanisms for multithreading support.

Further evidence that lazy threads are efficient is presented in Table 2, where we compare our lazy thread model with TAM for some larger programs on the Sparc. At this time our Id90 compiler uses a primitive version of explicit seed creation. In addition to the primitives described so far, the compiler uses strands, a mechanism to support fine-grained parallelism within a thread [9]. We see a performance improvement ranging from 1.1 times faster for coarse-grained programs like blocked matrix multiply (MMT) to 2.7 times faster for finer-grained programs. We expect an additional benefit of up to 30% when the compiler generates code using synchronizers.

Next, we look at the efficiency of work-stealing combined with seeds on a parallel machine by examining the performance of the synthetic benchmark proposed in [16] and also used in [25]. Grain is a doubly recursive program that computes a sum, but each leaf executes a loop of g instructions, allowing us to control the granularity of the leaf nodes. We compare its efficiency to the sequential C code compiled by gcc. As shown in Figure 9, using stacklets we achieve over 90% efficiency when the grain size is as little as 400 cycles. Compare this to the grain size of an invocation of fib, which is approximately 30 cycles. Most of the inefficiency comes from the need to poll the CM-5 network. The speed-up curve in Figure 9 shows that even for very fine-grained programs, the thread seeds successfully exploit the entire machine.

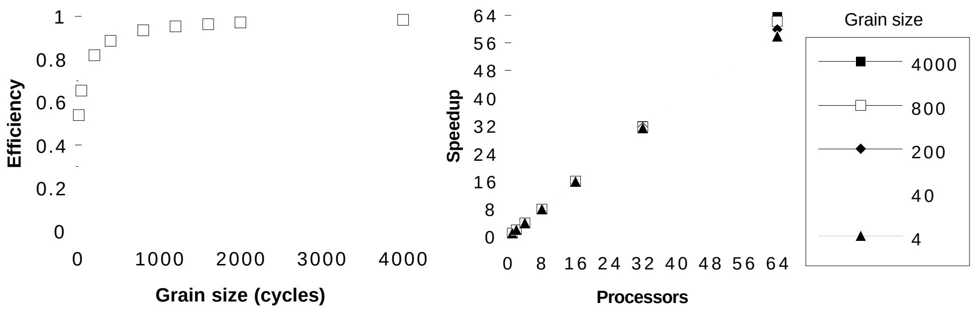


Figure 9 Efficiency of lazy threads on the CM-5 compared to the sequential C implementation as a function of granularity. We use the synthetic benchmark Grain [16, 25].

7 SUMMARY

We have shown that by integrating a set of innovative techniques for call frame management, call/return linkage, and thread generation we can provide a fast parallel call which obtains nearly the full efficiency of a sequential call when the child thread executes locally and runs to completion without suspension. This occurs frequently with aggressively parallel languages such as Id90, as well as more conservative languages such as C with parallel calls.

The central idea is to pay for what you use. Thus, a local fork is performed essentially as a sequential call, with the attendant efficient stack management and direct transfer of control and data. The only preparation for parallelism is the use of bounded-size stacklets and the provision of multiple return entry points in the parent. If the child actually suspends before completion, control is returned to the parent so that it can take appropriate action. Similarly, remote work is generated lazily. When a thread has work that can be performed remotely, it exposes an entry point, called a thread seed, that will produce the remote work on demand. If the work ends up being performed locally, it is simply inlined into the local thread of control as a sequential call. We exploit the one bit of flexibility in the sequential call, the indirect jump on return, to provide very fast synchronization and to avoid explicit checking for special cases, such as stacklet underflow.

Empirical studies with a parallel extension to C show that these techniques offer very good parallel performance and support fine-grained parallelism even on a distributed memory machine. Integrating these methods into a prototype compiler for Id90 results, depending on the frequency of parallel calls in the program, in an improvement by nearly a factor of two over previous approaches.

Acknowledgements

We are grateful to the anonymous referees and participants at the workshop for their valuable comments. We would also like to thank Manuel Faehndrich, Urs Hölzle, Martin Rinard, Nathan Tawil, and Deborah Weisser for their comments. Computational support at Berkeley was provided by the NSF Infrastructure Grant number CDA-8722788. Seth Copen Goldstein is supported by an AT&T Graduate Fellowship. Klaus Erik Schauser received research support from the Department of Computer Science at UCSB. David Culler is supported by an NSF Presidential Faculty Fellowship CCR-9253705 and LLNL Grant UCB-ERL-92/172.

REFERENCES

[1] A. W. Appel. *Compiling with continuations*. Cambridge University Press, New York, 1992.

[2] Arvind and D. E. Culler. Dataflow architectures. In *Annual Reviews in Computer Science*, volume 1, pages 225–253. Annual Reviews Inc., Palo Alto, CA, 1986.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, P. Lisiecki, K. H. Randall, A. Shaw, and Y. Zhou. *Cilk 1.1 reference manual*. MIT Lab for Comp. Sci., 545 Technology Square, Cambridge, MA 02139, September 1994.

[4] M.C. Carlisle, A. Rogers, J.H. Reppy, and L.J. Hendren. Early experiences with Olden (parallel programming). In *Languages and Compilers for Parallel Computing. 6th International Workshop Proceedings*, pages 1–20. Springer-Verlag, 1994.

[5] K.M. Chandy and C. Kesselman. Compositional C++: compositional parallel programming. In *Languages and Compilers for Parallel Computing. 5th International Workshop Proceedings*, pages 124–44. Springer-Verlag, 1993.

[6] E. C. Cooper and R. P. Draves. C-Threads. Technical Report CMU-CS-88-154, Carnegie-Mellon University, February 1988.

[7] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. TAM — a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, July 1993.

[8] J.E. Faust and H.M. Levy. The performance of an object-oriented threads package. In *SIGPLAN Notices*, pages 278–88, Oct. 1990.

[9] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy Threads, Stacklets, and Synchronizers: Enabling primitives for compiling parallel languages. Technical report, University of California at Berkeley, 1995.

[10] R. Hieb, R. Kent Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In *SIGPLAN Notices*, pages 66–77, June 1990.

[11] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *SIGPLAN Notices*, pages 32–43, July 1992.

[12] P. Hudak, S. Peyton Jones, P. Walder, B. Boutel, J. Fairbairn, J. Fasel, M.M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming language Haskell: a non-strict, purely functional language (version 1.2). *SIGPLAN Notices*, vol.27 (no.5): Ri–Rx, Rl–R163, May 1992.

[13] H. F. Jordan. Performance measurement on HEP — a pipelined MIMD computer. In *Proc. of the 10th Annual Int. Symp. on Comp. Arch.*, Stockholm, Sweden, June 1983.

[14] L.V. Kale and S. Krishnan. CHARM++: a portable concurrent object oriented system based on C++. In *SIGPLAN Notices*, pages 91–108, Oct. 1993.

[15] D.A. Kranz, R.H. Halstead Jr., and E. Mohr. Mul-T: a high-performance parallel Lisp. In *SIGPLAN Notices*, pages 81–90, July 1989.

[16] E. Mohr, D.A. Kranz, and R.H. Halstead Jr. Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, vol.2 (no.3): 264–80, July 1991.

[17] R. S. Nikhil. Id (version 88.0) reference manual. Technical Report CSG Memo 284, MIT Lab for Comp. Sci., March 1988.

[18] Rishiyur S. Nikhil. Cid: A parallel, "shared memory" C for distributed-memory machines. In *Languages and Compilers for Parallel Computing. 7th International Workshop Proceedings*. Springer-Verlag, 1995.

[19] R.S. Nikhil. A multithreaded implementation of Id using P-RISC graphs. In *Languages and Compilers for Parallel Computing. 6th International Workshop Proceedings*, pages 390–405. Springer-Verlag, 1994.

[20] A. Rogers, J. Reppy, and L. Hendren. Supporting SPMD execution for dynamic data structures. In *Languages and Compilers for Parallel Computing. 5th International Workshop Proceedings*, pages 192–207. Springer-Verlag, 1993.

[21] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the CHORUS distributed operating system. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–69. USENIX Assoc, 1992.

[22] David M. Ungar. *The design and evaluation of a high performance Smalltalk system*. ACM distinguished dissertations. MIT Press, 1987.

[23] M.T. Vandevoorde and E.S. Roberts. WorkCrews: an abstraction for controlling parallelism. *International Journal of Parallel Programming*, vol.17 (no.4): 347–66, Aug. 1988.

[24] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[25] D.B. Wagner and B.G. Calder. Leapfrogging: a portable technique for implementing efficient futures. In *SIGPLAN Notices*, pages 208–17, July 1993.

[26] Akinori Yonezawa. *ABCL– an object-oriented concurrent system* . MIT Press series in computer systems. MIT Press, 1990.