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. This paper presents an efficient execution framework for running these kinds of languages.

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 most other cases 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.

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 logically 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 3*(sect-stacklets)*, we address storage allocation. Since threads can fork other threads and each requires a stack, a tree of stacks, a *cactus stack*, is

[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.

figs/cactus.eps

Figure 1: *(fig-cactus)*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.)

required. We realize this cactus stack using *stacklets* (see Figure 1*(fig-cactus)*). 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 4*(sect-thread)*, 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 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 5*(sect-synchs)*, 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 6*(sect-seeds)*, 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 7*(sect-results)* 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 Motivation

The execution model that underlies many modern parallel programming languages depends on the dynamic creation of many independent (and often short lived) threads of control. The execution model is used either because it is made manifest in the language (e.g., futures, explicit forks, active objects, etc.), because the semantics of the language require it (e.g., non-strict evaluation) or because it facilitates compilation (e.g., ?). The use made of the threads demands that they be cheap and easy to create and manipulate. In this paper we present data structures and compilation techniques that reduce the overhead for thread creation and manipulation sufficiently to allow an unfettered use of multiple threads.

The languages described above all share common traits that make previous multi-threading packages unattractive. Often a thread will have the lifetime of only a single function call. In order to support dynamic load balancing threads may need to be migrated from one processor to another. They will often be employed to provide latency tolerance. These characteristics and others require thread creation and switching times on the order of a sequential call. Thread state must be small if communication time is not to overwhelm the advantages of migration. Clearly, POSIX style threads which work well in environments where a small number of long lived threads are created do not fit these requirements.

[Note: Example languages and the reason for having multithreading. Id90, pH, CC++, Charm++, Actor based languages, multi-T, Cilk, Cid, Olden, etc.]

Since threads can spawn other threads and each requires its own stack these systems require a tree of stacks as their logical storage model. This logical model is valid whether the new threads are created with recursive forks, parallel loops, futures, etc. Where the computation unfolds sequentially a stack of activation frames is sufficient. When a thread is spawned off, a new sequential stack is created and linked to the stack frame of the spawning thread. We call this particular form of tree of stacks a *cactus stack*. Whether parallelism is created by futures in multi-T, functions in Id90, method invocation on active objects in ABCL, explicit forks in Cilk, etc, the cactus stack fulfills the requirements.

Another common trait to these languages is that there is usually more logical parallelism in the application than physical processors. If the overhead of creating the excess parallelism if low enough it can be beneficial for both load balancing and latency tolerance. What is clear is that it all the logical parallelism does not necessarily need to be created in order to run the application. We define a *potentially parallel call* (or *ppcall*) to be a point in the computation where an independent thread can be created. There are two cases that demand that a ppcall be executed in parallel. First, in order for progress to be made, a thread may need to suspend and let another thread run. Second, to generate enough parallelism to eliminate processor idle time. If neither of these contingencies apply a ppcall could be executed as a sequential call.

In this paper we build a framework which allows us to eliminate the overhead of creating truly parallel threads except when necessary and thereby increasing the ease of compiling parallel languages and writing applications. Instead of executing ppcalls in parallel we execute them as sequential calls with the important feature that if they can at any time be elevated to independent threads of control if necessary, i.e., if they need to suspend.

We integrate the threading model into the compiler allowing us to tailor the thread creation and manipulation code to the context of its use. This stands in contrast to routines traditional thread libraries which must be general purpose. Additionally, this allows us to compile directly to a cactus stack.

2 Implementing Potential Parallelism

This section describes a spectrum of implementations for a potentially parallel call (*ppcall*) that can be used by the compiler to implement excess parallelism with the lowest possible overhead without restricting the amount of logical parallelism available to the application. In short, the compiler chooses for every ppcall the least expensive implementations of the call that meets the semantic requirements of that particular potentially parallel call.

There are two reasons a ppcall needs to run in parallel. First, to generate parallelism on demand. Second, to handle the need for a child task to suspend. If we could predict that a child would not need to suspend and that all the other processors will remain busy we could execute the child as a sequential call in the parent's thread. Clearly the execution with the lowest overhead (modulo locality and storage issues[2]) would execute ppcalls that need to run in parallel in parallel ab initio and would execute the remaining as sequential calls.

Since we do not know ahead of time which ppcalls need to run in parallel we must pay some cost for representing the ppcall such that it will run in parallel only if necessary. Our goal is to minimize this overhead without reducing the amount of parallelism available to the application. To fulfill this goal we define a hierarchy of implementations for the ppcall. The most expensive, given that we predict most ppcalls to run sequentially, is the explicit parallel call. The cheapest is the parallel-ready

[2]We ignore the fact that some execution paths could lead to blow up in the amount of frame storage required by the program.

sequential call; which behaves like a sequential call unless the called child suspends, whereupon it transforms itself into a parallel call in its own thread.

In between these two extremes are representations of the ppcall that can be instantiated as either parallel or parallel-ready sequential calls depending on the needs of the application. A *full closure* is used when the arguments to the ppcall may be changed by the parent before the closure is activated. It is put on an explicit queue when created. An *explicit seed* is essentially a specialized closure that does not need to store the arguments to the ppcall. Instead it points to code that executes in the context of the parent which instantiates the ppcall. The *implicit seed* is even more specialized than the explicit seed because it is not put on an explicit queue; instead it is linked in the activation frame of the parent.

These last three implementations of ppcall are nascent threads (or, if run sequentially, nascent activation frames). Because the cost of creating them is nominal (or even free), they allow us to produce parallelism on demand without the overhead normally associated with parallel calls. When parallelism is not needed we activate these nascant threads as parallel-ready–sequential calls; which allows us to elevate a sequential child into an independent concurrent thread. In short, the run time structures that we introduce allow us to implement ppcall with only nominal overhead without reducing available parallelism.

2.1 Requirements of ppcall

A ppcall must provide the same semantics of a parallel call even if the ppcall is run like a sequential call in the same thread as its parent. We identify three independent actions that occur for a parallel call. First, the child must be able to return results to the parent so that the parent can use them in the future. We call the code that performs this task an *inlet*. Second, the parent must update the synchronization status when the call is issued and then when it returns. Finally, a continuation address (contadr) must be maintained in the parent to indicate what actions are needed to be performed after all of its children have completed. The contadr is the location of the next instruction to be executed in the parent after the ppcall begins.

A sequential call combines these three actions into a single address, the return address, and the function call interface. The return address points to the inlet. No synchronization state is needed since the parent only executes when the child has completed. Following the inlet code is the code for the rest of the function, i.e., the return address is also the contadr.

It is our goal to reduce the overhead of ppcalls to that of sequential calls when the logical parallelism represented by the ppcall is not needed. In order to meet this goal, we must combine the above three independent actions such that we can still elevate a sequentially invoked ppcall into its own thread. Suppose there are n consecutive ppcalls followed by a join. The main difficulty is that at the point of invoking any but the last ppcall the contadr can no longer point to the join, but must point to the ppcall that follows the one being invoked. Indeed, if the ppcall executes sequentially the return address can work as does the sequential call. However, if the child suspends, or some figs/states.idraw

Figure 2: *(fig-states)*The legal state transitions for a task.

other ppcall is started while the child is still running (e.g., some other processor stealing work), the child must now have an independent inletadr and the contadr in the parent will have to be updated.

Clearly the above argument shows us that we cannot use a sequential call to represent a ppcall. One can think of many ways to begin a ppcall as a sequential call and then if necessary convert it into a parallel call. All the methods will have to create a new thread (for either the suspended call or its parent) and turn the original return address into both an inlet address and a contadr, update the synchronization state of the parent, and finally modify the parent to account for the fact that its child is now a parallel task. This general method can be used both for children that suspend and children that are migrated to another processor.[3]

2.2 The Abstract Machine

Before describing the variety of representations for the ppcall we need to define the abstract machine that underlyes our control model. Every child of a ppcall is considered a logically separate task that can run in its own thread. There is no preemption. We do not need to worry about fairness since for the entire application to finish all the tasks will need to run to completion.

Each task is identified by its activation frame (or frame), analogous to a stack frame in a sequential language. A frame can be in one of four states: running, ready, idle, and nascent. There is exactly one running frame per processor. A ready frame is a frame that the scheduler may run when the running frame becomes idle. An idle frame is waiting for some event to become a ready frame. A nascent frame is not truly a frame, but some representation of the task that is ready to become a frame. The state transitions for a frame are shown in Figure 2*(fig-states)*.

In our system, only nascent frames may be migrated between processors. We choose not to migrate already created frames for a variety of reasons. If we allow full fledged frames to migrate we would have to choose to either limit the execution model to shared memory architectures, forbid pointers into stack frames, or make all pointers global pointers incurring an unnecessary overhead. Further, in many cases migrating an already created frame, can lead to poor data locality.

[3]For reasons discussed in Section **??***(sect-migrate)* we choose not to do migration of any tasks once that have begun execution.

Frames give up the processor in one of three ways: they call a potentially parallel child, they suspend, or they return to their parent. The first and last of these mimic the way in which control transfer is handled in sequential stack-based languages. We choose to follow this control transfer, because it allows us to use registers for transferring arguments between parent and child, and for transferring results between the child and the parent. Furthermore, in this way if a ppcall executes sequentially we expect to get the same performance from the ppcall as we would from a sequential call. Thus, when a frame invokes a child it is in some sense making the state transition from running to ready. Instead of putting the parent on an explicit ready queue, we use the activation frames as an implicit ready queue.

When a frame suspends there are four possible choices for the next frame to run: its parent frame, a frame from the ready queue, a nascent frame, an explicitly selected frame. Clearly the choice of which frame to run has a significant impact on the overall performance of the system and the amount of resources consumed by the application. The default action is to run the parent. If the child chooses to run another frame, it must put its parent on an explicit ready queue.

The transition from nascent to running can occur whenever a nascent frame is activated, or in other words, when the representation of the ppcall is invoked. From ready to running when a suspended frame explicitly schedules the frame, when a child returns or suspends and transfers control it parent, or when the system scheduler picks a frame from the ready queue. The transition from running to idle happens when a frame suspends. Finally, the transition from idle to ready occurs when an inlet runs and enables the frame, posting it onto an explicit ready queue.

We also impose an implied scheduling order to the forks in a forkset. These forks are spawned off (with whatever representation is chosen for them) in the order in which they appear lexically. Thus, if they do run sequentially, the first starts, the it returns, followed by the second staring, etc. They follow this order even if a child suspends. Thus, When the i^{th} child suspends (or returns) the $i + 1^{th}$ child will be invoked next. If the i^{th} child chooses not to return to the parent, then it must enqueue the parent on an explicit queue so that the $i + 1^{th}$ (and all the rest) of the child can be called. [4]

2.3 The cost of a ppcall

We define the overhead cost of a representation as the extra cost it introduces above that of a sequential call. Since any of these calls may end up executing in parallel we define a secondary cost, the transformation cost, as the cost of running the call in parallel versus having executed it in parallel ab initio. As we shall see, most of these representations have very small transformation costs, while their basic cost varies from zero to the cost of an explicit parallel call (minus the base cost of a sequential call).

There are three aspects that make up the cost of a representation: creation, linkage, and instantiation. Creation refers to the cost of creating the nascent form of the call

[4] should I make explicit the fact that if the 1st child suspends, then it can't return unless all the other children have been invoked? If this is not true, then the parent will be on a queue somewhere

figs/lattice.idraw

Figure 3: *(fig-lattice)*The hierarchies of costs and generality for ppcall implementations.

which can later be instantiated into an executing call. Linkage refers to the cost of enqueing the representation so that it can later be found and executed. The instantiation phase represents the cost of turning the representation into an executing call. Clearly an eager explicit parallel call and the sequential call have no creation or linkage costs. For these two representations the instantiation phase is the cost of either creating a new thread or a new sequential child. These costs depend on the underlying runtime structures used to represent threads, stacks, etc. and are discussed in the sections following this one.

The transformation cost has two components: direct and indirect. The direct part is the cost associated with converting the sequential call into a parallel call. The indirect component is the cost that the conversion adds to the parent of the call being converted (and the parent's future children).

2.4 The ppcall hierarchy

The hierarchy of implementations of ppcalls spans the spectrum from the parallel ready sequential call to eager parallel call. In between these two extremes we find, implicit seeds, explicit seeds, and the general closure. Each of these implementations increases in cost and adds more generality. (See Figure 3*(fig-lattice)*.) The job of the compiler is to choose the cheapest call that is sufficient to the context of the ppcall.

The parallel ready sequential call has minimal overhead cost. In fact, by definition it is the cheapest method that can be used to implement a ppcall without other additional knowledge. Like the sequential call this implementation has no creation or linkage cost. Using the techniques described in Section **??***(sect-prsc)* the execution cost is only slightly higher than that of a sequential call. The direct transformation cost is zero in that it costs the same to suspend a parallel thread as a parallel ready sequential call. The indirect cost depends on the frame storage method used and will be discussed in Section **??***(sect-frame-store)*.

The parallel ready sequential call lacks generality since it eliminates parallelism when it is used.[5] The child begins execution in the parent's thread. For example, if we represent the forks of an n-way fork as prs call, then how do we find the parallelism present in the forks? For this we must us a more general representation.

The next representation up the hierarchy in both generality and overhead cost is the implicit seed. The implicit seed is essentially a triple of continuations, any of which when execute will instantiate the work it represents. The triple represent three different possible execution scenarios: sequential, concurrent, and parallel. The sequential execution scenario occurs when the implicit seed is instantiated as a parallel ready sequential call. The concurrent execution occurs when one of the other children in the fork-set suspended and multiple threads of control are needed to complete the fork-set. Parallel execution occurs when parallelism is needed for load-balancing and the call is to be executed on another processor.

The implicit seed may only be used for a ppcall that follows another ppcall lexically in the same fork-set. This constraint arises because we use the return address of the preceding call as the implicit seed. In other words, we extend the concept of the return address to include the triple needed for the implicit seed (see Section **??***(sect-implicit-seed)*). The creation cost of this representation is only the cost of creating the continuation triple. Using return vectors (see sectret-vector) this cost is zero. Since the return address must always be saved, the linkage overhead for an implicit seed is zero.

Implicit seeds work very well in handling the problem of suspension. When a sequentially called ppcall suspends, it uses the return address it has to locate the implicit seed, and then executes the concurrent continuation address. However, if a remote processor wants to execute the parallel continuation it will have to (in the general case) do a tree walk of all outstanding frames to find an implicit seed. The result is that direct transformation cost is zero when the reason is that a previous child suspended and o(number of frames) when a remote processor wants work.

Explicit seeds allow remote processors to find the seed in constant time. They are also a continuation triple, but instead of using the preceding call's return address for linkage, they are pushed onto a queue. Thus they have zero creation cost, but a small linkage cost—the cost of putting them on the queue and later removing them if the ppcall they represent ends up executing sequentially. However, the direct transformation cost is that of a dequeue independent of the reason.

They are additionally more flexible than implicit seeds since they are not constrained by the need for using the preceding call's return address. As long as the arguments to the ppcall are not modified, a seed for the ppcall can be created and enqueued at any point in the parent.

If more generality is needed, the closure representation is used. The closure representation creates a closure of the callee and any arguments it needs are packaged up in a data structure at creation time. The closure is then enqueued on some scheduling queue. The closure has a creation cost since if it is executed sequentially the creation of closure was unnecessary as was the enqueing and dequeueing of the closure. The transformation cost is also non-zero since if it were initially executed in parallel we also would

[5] For reasons given later we reject migration as a means of generating parallelism.

not have needed to create the closure. However, the overhead cost is low compared to an eagerly executed parallel call that could have been executed sequentially.

2.5 Choosing the representation

The ppcall hierarchy described here depends on being implemented not in a library but as part of the compilation process. Each representation depends on the details of the call/return interface, the scope of the fork-set, etc. In this section we describe which representations would be used for four basic patterns of ppcall usage: loop based, n-way forks, 2-way forks, and 1+delta-way forks. We believe that these patterns cover the important needs of modern parallel languages that are based on task based parallelism.

loop based - todo

The canonical pattern for task parallelism is a forkset with n forks followed by a join, an n-way fork. This pattern can be compiled using prs call for the first fork and either an implicit or explicit seed for the following $n - 1$ forks. We don't need to use a closure for the subsequent forks because the seed will only be executed in the context of the parent; which already has storage for the arguments. The choice of an implicit or explicit seed will affect the ease of performing work stealing. On a single processor system an implicit seed would be sufficient.

We distinguish the special case of n-way forks when $n = 2$. Like n-way forks, the first is represented as a prs call and the second as a seed. It is a special case because we can use the synchronizer techniques described in Section **??***(sect-synchronier)* to eliminate the need for explicit synchronization. Though, synchronizers can be used for n-way forks when $n > 2$, it is not cost effective because the synchronizer technique in effect combines the inlet address, contadr, and synchronization state and the result is that n^2 synchronizers are needed for an n-way fork.

The last pattern we recognize is the $1 + \delta -$ wayfork. This occurs when a fork is followed by sequential code in the parent which is followed by a join. Figure **??***(fig-oned)* shows an example of this pattern. In this case if we used a prs call to represent the fork, we would lose the parallelism in the function. If we allowed migration we could use a prs call, and then the parent could be migrated. However, it seems that this pattern is used mostly when the code following the fork is to execute on some local data. If migration were used the fork would be executed locally and the local code would be executed remotely.[6] Instead, we represent the first fork as either a seed, or if the arguments to the fork are possibly modified by the following code as a closure. We enqueue the seed or closure and start executing the following code. If no other processor stole the ppcall, then when the join was reached the parent would suspend and the ppcall would be executed.

[6] Another approach would have the compiler package up the following code as a separate function and use the 2-way fork pattern. But this suffers from the same problems as migration.

2.6 Discussion

compare migration to our system. discuss combining the inletadr, contadr, and synch show how compiler is necessary. discuss implicit v. explicit seeds and the conversion from one to other

3 Storage Management: Stacklets

(sect-stacklets)

Analogous to the hierarchy of representations of the ppcall, we also introduce a hierarchy of frame stores. Depending on the type of call and the requirements of the child the frame can be stored on either a stack, stacklet or heap. This hierarchy of frame stores allows us to choose the least expensive storage mechanism for the activation frame.

The least versatile and least expensive method is to use the stack. This is only available for purely sequential calls, i.e., for calls that can never suspend. The most expensive method is to store each frame in a heap allocated memory segment. Using a heap allocated frame store requires links to be created from child to parent—something that is implicit when using the stack.

Between these two extremes we use stacklets for frames that are allocated for ppcalls. A stacklet combines the efficiency of a stack for frames that may become or are independent threads of control. This method allows us to assign to each thread its own logically unbounded stack without having to manage multiple stacks or to require garbage collection. In the rest of this section we describe the mechanics of stacklets and discuss there advantages over other intermediate methods of storing activation frames.

Our compilation strategy allows us to use the sequential stack for sequential calls, stacklets for small to moderate sized ppcalls, and the heap for large ppcalls.

3.1 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 Figure 4*(fig-stacklet)*). 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 5*(fig-seq-alloc-no-overflow)*). If not, figs/stacklet.idraw

Figure 4: *(fig-stacklet)*The basic form of a stacklet.

figs/seq-alloc-no-overflow.idraw

Figure 5: *(fig-seq-alloc-no-overflow)*The result of a sequential call which does not overflow the stacklet.

figs/seq-alloc-overflow.idraw

Figure 6: *(fig-seq-alloc-overflow)*The result of a fork or of a sequential call which overflows the stacklet.

a new stacklet is allocated and the child frame is allocated on the new stacklet (see Figure 6*(fig-seq-alloc-overflow)*). This also happens for a fork, which causes a new stacklet to be created on the local processor. We could 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 6*(fig-seq-alloc-overflow)*). 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 7*(fig-remote-alloc)*).

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 figs/remote-alloc.idraw

Figure 7: *(fig-remote-alloc)*A remote fork leaves the current stacklet unchanged and allocates a new stacklet on another processor.

stacklet overflows, a new stacklet is allocated from the heap. This cost is amortized over the many invocations that will run in the stacklet.

3.2 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.

3.3 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.

3.4 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.

We tested out stacklets by comparing the doubly recursive fib function implemented with stacklets versus a hand optimized C version that does not use the Sparc register windows. Fib, being fine-grained, is a good "stress test" of function invocation. The result is that manipulating stacklets yields a 14% slowdown.[7]

The main disadvantage in using stacklets is that if a frame is at the top of a stacklet, each of its children will have to be allocated on a new stacklet. This boundary case is similar to the overflow case when using register windows. We can reduce the possibility of this happening in two ways. First, we can increase the size of each stacklet so that it happens rarely. Second, if a function makes many function calls the compiler can check to see that the parent and the maximum sized child can both fit in the stacklet. If both will not fit, then the parent will allocate a new stacklet guaranteeing that the boundary condition does not occur.

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

[7] We had to use an assemble version of fib because the native GCC optimized fib, using stacklets actually runs 12% slower than the stacklets version.