LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Indolent Closure Creation

Technical Memo
MIT-LCS-TM-580
June 24, 1998

Volker Strumpen
strumpen@supertech.lcs.mit.edu

This manuscript contains extended notes of a talk entitled "Indolent Closure Creation," held by the author at the Yale Multithreaded Programming Workshop in New Haven, June 8–9, 1998.

This manuscript describes research pursued at the Laboratory for Computer Science of the Massachusetts Institute of Technology as part of the Cilk Project. Funding for this work has been provided in part by DARPA Grant N00014-94-1-0985.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Indolent Closure Creation

Volker Strumpen*
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
strumpen@supertech.lcs.mit.edu

June 24, 1998

Abstract

A ***closure*** is a representation of a thread in memory, ready to be executed. The goal of this work is to create *portable* closures that can be transferred across binary incompatible architectures. Consequently, indolent closures are software-implemented, and rely on a copy mechanism which allows for potential data representation conversion on-the-fly. Indolent closure creation optimizes the current implementation of the algorithmic multithreaded language Cilk. Rather than generating parallelism eagerly by pushing closures on a ready deque whenever a parallel spawn occurs, indolent closures are created only if the ready deque is empty and a steal request occurs. Indolent closure creation is based on the *release-and-resume transformation*, developed for the portable checkpoint compiler porch. Experimental results on various architectures show that indolent closure creation reduces the cost of a parallel spawn to be competitive with the cost of a sequential function call.

Keywords: Cilk, Indolent Closure Creation, Lazy Threads, Multithreading, Porch, Portable Checkpoints, Work-First Principle.

1 Introduction

The success of an abstraction depends on the efficiency of its implementation. Multithreading is a powerful abstraction for parallel systems. This memo proposes an optimization for a key issue in implementing portable multithreaded languages efficiently: How to minimize the overhead of generating parallelism.

The question whether an abstraction can be implemented efficiently has been subject to flamy appeals in the past. In 1977, Steele [17, p. 16] advocated the *procedure call*: "Procedure calls are demonstrably not inherently as inefficient as computing folklore would lead us to believe. There are implementations of higher-level programming languages in which procedure calls are almost as cheap as GOTO statements." Today, procedure optimizations such as function inlining, leaf-routine optimization and tail-recursion elimination are ubiquitous in modern compilers, and structured programming has prevailed. When garbage collection became popular, the well-established stack abstraction has been questioned by Appel in an article [2] entitled "*Garbage Collection Can Be Fater Than Stack Allocation*." In response, Miller and Rozas [10] claimed that "*Garbage Collection is Fast, But a Stack is Faster*."

*This manuscript is an extended version of a talk entitled "Indolent Closure Creation", which I gave at the Yale Multithreaded Programming Workshop in New Haven, June 8–9, 1998.

Implementing multithreading in an efficient manner has lead to a variety of proposed multithreaded architectures [1, 6, 16, 19]. Efficient software implementations have been proposed for commodity architectures that introduce a compromise between stack and heap allocation techniques. For example, Lazy Threads [8] are based on *stacklets*, and the Illinois Concert system [9] employs a *hybrid stack-heap execution mechanism*. The multithreaded Cilk language [7] exhibits a striking balance between versatility of threads, portability, and efficiency. Cilk's implementation is based on the *cactus stack* semantics proposed by Moses in 1970 [12]. As operating systems and compilers are turning into commodity components, it is desirable to implement multithreading in a portable manner, that is independent of processor architecture, operating system and compiler. In this memo I propose a portable software-implementation of threads with a parallel spawn that is almost as efficient as a function call.

Throughout this manuscript the term ***closure*** refers to the representation of a thread in memory, ready to be executed. The term ***activation frame*** denotes the private memory area of a function on the runtime stack, which holds local variables and parameters among other data. The terms ***activation record*** or ***stack frame*** are often used synonymously for activation frame. A closure stores slightly more information than an activation frame, which enables threads to resume execution from a closure independent of the parent thread or spawned children, and to migrate closures across processors. A comprehensive discussion of the information necessary to be stored in a closure can be found in [8].

Two design goals guided the work presented in this memo:

1. We are interested in two types of ***portability***:
   (a) The code for manipulating activation frames on the runtime stack should be independent of a particular architecture, operating system, and compiler to enable simple installation of the multithreaded language.
   (b) The representation of closures should be machine independent to allow for transferring closures across binary incompatible architectures.
2. We want to minimize the ***cost of closure creation***, such that it permits efficient execution of fine-grained threads. Typically, automatically parallelizing compilers can only coalesce fine-grained threads efficiently. Such fine-grained threads must be created with minimal overhead at runtime in order to execute efficiently.

Portability excludes implementations that modify a compiler in order to provide a customized stack layout, and customized manipulation of the stack pointer and frame pointer, as done for Lazy Threads [8]. Instead, indolent closures rely on source-to-source compilation to enable machine independence. We deploy the porch compiler technology for portable checkpointing [18] to that end.

In general, coarse-grained threads do not require a very fast creation mechanism, because the execution time of coarse-grained threads will amortize the cost of thread creation. Indolent closure creation has been investigated in the context of the Cilk multithreaded language. We require a multithreaded computation to be structured into fine-grained threads, because we do not want to sacrifice the guarantees of Cilk's randomized work-stealing scheduler. Under this constraint, indolent closure creation should not affect the performance of a multithreaded computation. Indolent closure creation minimizes the cost of generating parallelism by reducing the cost of a `spawn` operation almost to the cost of a sequential function call.

The remainder of this memo is organized as follows. Section 2 discusses the implementation of a parallel spawn in Cilk and the work-first principle. Section 3 describes the portable checkpoint compiler porch. In particular, one of the source-to-source transformation of porch, the *release-and-resume transformation*, renders the stack environment portable. This technique is the basis of indolent closure creation, which is introduced in Section 4. The experimental results presented in Section 5 show that the cost of a parallel spawn can be reduced almost to the cost of a sequential function call. Related work is sketched in Section 6, and conclusions for future work are drawn in Section 7.

2 Cilk and the Work-First Principle

The idea of indolent closures has been motivated by the work-first principle that underlies the implementation of the Cilk language. Cilk extends the C programming language [7] into an algorithmic multithreaded language. In Cilk, parallelism is exposed explicitly by means of the `spawn` and `sync` keywords. If the `spawn` keyword precedes the call of a procedure, the calling parent can continue execution in parallel with the called child. If a Cilk program is executed on one processor, the semantics of the parallel `spawn` is equivalent to that of a conventional function call. A statement containing the `sync` keyword acts like a local barrier to synchronize the parent with its spawned children. Cilk's randomized work-stealing scheduler [4] schedules the parent and its spawned children across the processors of a parallel machine in a provably efficient manner. An idling processor becomes a ***thief*** that picks a ***victim*** processor at random and attempts to steal work.

A Cilk computation is characterized by its ***work*** and its ***critical-path length***. Figure 1 shows the representation of a Cilk computation as a directed acyclic graph (DAG). Work T_1 is the total execution time of the sequential execution of a computation, in the example $T_1 = 10$. The critical-path length T_∞ is the execution time of a computation on an infinite number of processors. The computation in Figure 1 has a critical-path length $T_\infty = 6$.

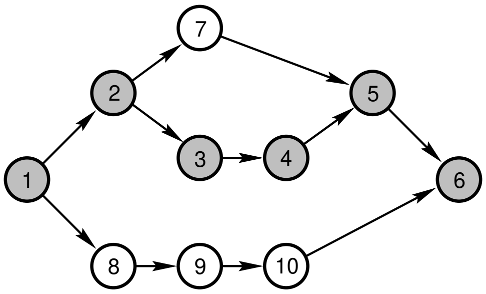

Figure 1: DAG representation of a Cilk computation. The critical path consists of the shaded nodes.

The ***parallel execution time*** T_P of a Cilk computation on P processors is bound by $T_P \geq T_1/P$ and $T_P \geq T_\infty$. The former lower bound holds, because at most P units of work can be executed in a single step of the computation. The latter lower bound is imposed by the structure of the computation, which requires at least the number of steps dictated by the critical path length. Blumofe and Leiserson [4] have shown that Cilk's randomized work-stealing scheduler executes a Cilk computation in expected time $T_P = T_1/P + O(T_\infty)$.

The work-first principle is based on the analysis of the expected parallel execution time, as described in [7]. An upper bound on T_P is given by the ***critical-path overhead***, which is defined as the smallest constant c_∞ such that $T_P \leq T_1/P + c_\infty T_\infty$. Furthermore, with the definitions of ***parallelism*** $\overline{P} = T_1/T_\infty$ and ***parallel slackness*** $\overline{P}/P$, the ***parallel slackness assumption*** $\overline{P}/P \gg c_\infty$ states that the number of processors P used to execute a multithreaded Cilk computation is much smaller than the parallelism $\overline{P}$. Under the parallel slackness assumption Cilk computations achieve linear speedup, because the parallel slackness assumption implies that $T_P \approx T_1/P$.

Any implementation of a program in Cilk incurs overhead compared to its sequential implementation in C, the so-called *C elision*, because parallelism requires creation and synchronization of threads. If the sequential execution time of a C program is T_S and the execution time of the corresponding Cilk program on one processor is T_1, the ***work overhead*** can be defined as $c_1 = T_1/T_S$. If parallel slackness is assumed, $T_P \approx c_1 T_S / P$. This approximation suggests the following formulation of the work-first principle:

Work-First Principle: Minimize the scheduling overhead borne by the work of a computation. Specifically, move overheads out of the work and onto the critical path. Formally, minimize c_1, even at the expense of a larger c_∞.

Any implementation of the Cilk language should in particular reduce the cost of spawning threads to a minimum, because the `spawn` operation contributes exponentially to the work of a computation but only linearly to the critical path. The current implementation of Cilk-5 is based on the work-first principle. The doubly-recursive implementation of a Fibonacci-number computation, shown in Figure 2, is used throughout this manuscript to illustrate the issues involved in optimizing closure creation and in particular the implementation of Cilk's `spawn` operation.

```
1  cilk int fib(int n)
2  {
3     if (n < 2)
4        return n;
5     else {
6        int x, y;
7        x = spawn fib(n-1);
8        y = spawn fib(n-2);
9        sync;
10       return x+y;
11    }
12 }
```

Figure 2: Cilk implementation of `fib`.

The key insight underlying Cilk's randomized work-stealing scheduler [4] is that the number of steals $O(PT_\infty)$ is substantially smaller than the number of `spawn` operations. This relatively small number of steals stems from the following observation: If the work stolen from a ***victim*** processor is close to the root of the DAG of a computation, the ***thief*** processor is likely to obtain a relatively large piece of work, which will keep the thief busy for a relatively large amount of time. Cilk can be implemented efficiently, because virtually all overheads due to parallelism can be amortized by shifting them into the implementation of the relatively few steal operations.

This strategic decision is a perfect match for stack-based runtime models that are exploited in imperative languages such a C. Figure 3 illustrates the work-stealing idea for the Fibonacci computation. Two processors are shown, a victim and a thief. Each processor owns a private ***runtime stack***, which is conventionally used to provide storage for the activation frames of each function called during execution. According to the runtime model of imperative languages, functions are executed in depth-first order, that is the execution of a function is started as soon as it is called. As a consequence, a new activation frame is pushed onto the runtime stack when a function is called, and popped from the runtime stack upon return. A Cilk computation executes parallel spawns like ordinary function calls unless a steal occurs.

The victim processor in Figure 3 is assumed to have executed a sequence of recursive function calls `fib(n-1)` (line 7 in Figure 2) until it reaches the base case `fib(1)`. Consequently, the runtime stack is populated with activation frames, starting with `fib(n)` at the bottom, and ending with `fib(1)` at the top. The Cilk language preserves the C semantics of function calls whenever a `spawn` is executed. There is a difference, however. Before actually calling the spawned procedure, a closure is pushed onto the ***ready***

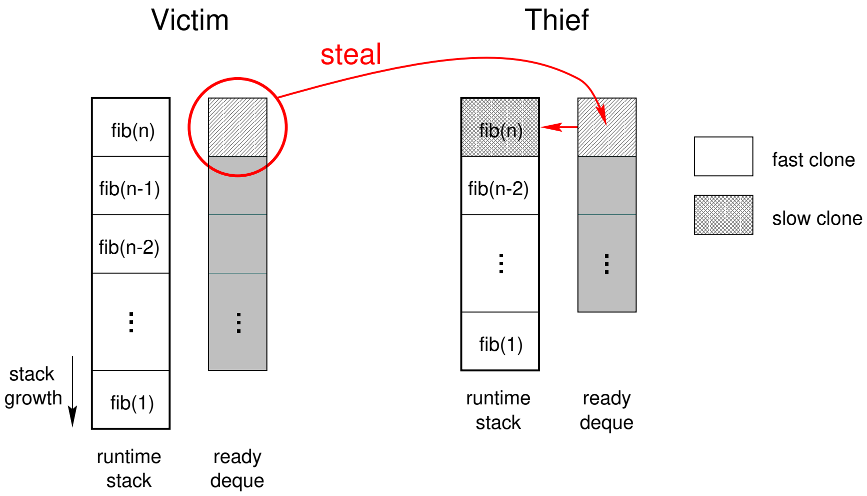

Figure 3: Stack handling in Cilk.

deque [7]. Figure 3 shows the ready deque of the victim, which holds closures for all activation frames corresponding to procedure invocations from `fib(n)` to `fib(2)`. The base case `fib(1)` does not spawn further invocations. Closures are drawn next to their activation frames. Among other data, they contain the live state of the activation frames and the program counter of the statement following the associated `spawn`.

At the point of execution where the victim executes `fib(1)`, the thief may execute a steal operation. The steal affects the closure at the bottom of the victim's ready deque, which is circled in Figure 3. After stealing the closure corresponding to `fib(n)`, the thief calls procedure `fib` and loads all live variables from the closure into its activation frame on the runtime stack.[1] Subsequently, the thief resumes execution at the statement *following* the `spawn` operation which saved the closure. In the example of Figure 3, the runtime stack contains the state of the computation when executing the first `spawn` operation in line 7 of Figure 2 recursively. Thus, the thief will resume execution of `fib(n)` in line 8 of Figure 2, that is it spawns `fib(n-2)`.

Note that stealing always affects the bottom closure of the ready deque in Cilk. Because the largest amount of work to be executed is generally represented by this closure, Cilk's work-stealing scheme incurs relatively few steals. Also, loading the contents of a closure into the corresponding activation frame on the runtime stack occurs only during the first procedure invocation of the thief after a steal. This fact has motivated a compilation scheme for Cilk, where two clones are generated for each Cilk procedure, a ***fast clone*** and a ***slow clone***. The fast clone is executed in the common case. It contains the code to push a closure onto the ready deque whenever a `spawn` operation is encountered. The slow clone, in contrast, is only executed on the thief processor after a successful steal. It is instrumented with code to load a closure into its corresponding runtime activation-frame.

According to the work-first principle, the fast clone should be implemented such that the overhead for managing parallelism is minimized. Nevertheless, the fast clone still suffers from a substantial performance penalty, although the current Cilk-5 implementation is based on the work-first principle. The reason is that closures are pushed onto the ready deque before every `spawn` operation. This *eager* provision of parallelism is the subject of the optimization described in this paper.

The fast Cilk-clone of `fib` is shown in Figure 4. It is generated from the Cilk source of the Fibonacci program shown in Figure 2. Code inserted by the compiler is shown in grey. The first `spawn` operation

[1] Loading live variables is only necessary if disjoint address spaces are spanned. If shared memory is available, loading the live variables reduces to indirecting the accesses to variables.

```
1  int fib(int n)
2  {
3     fib_frame *f;
4     f = alloc(sizeof(*f));
5     f->sig = fib_sig;
6     if (n < 2) {
7        free(f, sizeof(*f));
8        return n;
9     }
10    else {
11       int x, y;
12       f->entry = 1;
13       f->n = n;
14       *T = f;
15       push();
16       x = fib(n-1);
17       if (pop(x) == FAILURE)
18        return 0;
19       y = spawn fib(n-2);     /* not transformed */
20       /* sync is nop */ ;
21       free(f, sizeof(*f));
22       return x+y;
23    }
24 }
```

Figure 4: Fast clone of the Cilk version of `fib`. Compiler-generated code is shown in grey.

(line 7 in Figure 2) is translated into lines 12–18. The code generated by the transformation of the second `spawn` operation (line 8 in Figure 2, line 19 in Figure 4) is omitted in Figure 4. The compiler-inserted code implements the management of Cilk closures. Whenever the fast clone is invoked, a new closure (`fib_frame`) is allocated (lines 3–5), and deallocated upon return (line 7 and line 21). During each `spawn` operation, the closure is updated by saving the portable representation of the program counter (line 12), saving the local live variables (line 13), and pushing the closure onto the ready deque (lines 14 and 15). Lines 17–18 contain the check whether the parent of the returning thread has been stolen.

This section described the eager provision of parallelism in Cilk. The commonly executed fast clone of a Cilk procedure creates closures on the ready deque upon visiting a parallel spawn. These closures are available for stealing. Indolent closure creation is an optimization of this process, which generates parallelism in a truly lazy fashion. Before introducing indolent closure creation, I describe the portable checkpoint compiler porch. The method for providing a portable stack environment employed by porch is the basic implementation technique underlying indolent closure creation.

3 Porch and Stack Environment Portability

The porch compiler [15, 18] is a source-to-source compiler that translates C programs into semantically equivalent C programs additionally capable of saving and recovering from portable checkpoints. ***Portable checkpoints*** capture the state of a computation in a machine-independent format, called ***Universal Checkpoint Format—UCF***. The code for saving and recovering as well as converting the state to and from *UCF* is generated automatically by porch.

The porch compiler technology solves three key technical problems to render checkpoints portable.

Stack environment portability: The stack environment is deeply embedded in a system, formed by hardware support, operating system and programming language design. The key design decision to implement porch as a source-to-source compiler has been due to the necessity to avoid coping with system-specific state such as program counter or stack layout. It is not clear whether this low-level system state could be converted across binary incompatible machines. Instead, porch generates machine-independent source code to save and recover from checkpoints. At the C language level, variables can be accessed by their name without worrying about low-level details such as register allocation or stack layout done by the native compiler.

Data representation conversion: Two issues of data representations are of concern: bit-level representations and data layout. Basic data types are stored in different formats at the bit level. The most prominant formats are *little endian* and *big endian*. Furthermore, (memory) system designs require different alignments of basic data types. These determine the layout of complex data types such as structures. Consequently, all basic data types and the layout of complex data types are translated into a machine-independent format. The porch compiler generates code to facilitate the corresponding conversions automatically.

Pointer portability: Pointers are rendered portable by translating them into machine-independent offsets within the portable checkpoint. Since the target address of a pointer is not known in general at compile-time, porch is supported by its runtime system to perform the pointer translation during checkpointing and recovery.

To enable code generation, ***potential checkpoint locations*** are identified in a C program by inserting a call to the library function `checkpoint()`. For these potential checkpoint locations, porch generates code to save and recover the computation's state from portable checkpoints.

In the following, we explain our technique for checkpointing and recovering the runtime stack in a portable manner. Source-to-source compilation enables portability. The key idea is to access all stack-allocated variables by their *names* when saving and recovering from checkpoints. Register allocation and stack layout are not an issue at the source-code level, where porch operates.

Accessing variables by means of their names requires entering their lexical scope. To checkpoint the local variables of functions on the runtime stack, we start with the currently active function on top of the stack, save its local variables, and recursively visit its caller function until the bottom of the stack is reached. During recovery, the process is reversed. A function frame is pushed onto the stack, and its local variables are loaded from the checkpoint. Then, the callee of the original call sequence is pushed onto the stack until the runtime stack is rebuilt.

The question is, how can the control flow be redirected during checkpointing to enable access to local variables by their names. The only portable mechanism to visit stack frames, available on every general purpose processor architecture, is the standard function call and return. This mechanism is henceforth used to instrument functions in order to provide for extraordinary returns, called ***releases***, and extraordinary calls, called ***resumes***. A stack frame is released by remembering the ***resume-point*** and returning (standard function return) to the caller. A stack frame is resumed by calling the function (standard function call), and jumping to the resume-point without executing any of the function's original statements. Two code constructs implement the release and resume functionalities, a "jump table" and "call wrappers."

Definition 1 A ***jump table*** implements a *computed goto*. It consists of a switch of *goto* statements, one of which is selected by the resume-point identifier.

Definition 2 A function ***call wrapper*** consists of two parts, the ***prologue*** and the ***epilogue***. The prologue consists of an assignment to a local state variable `callid`, which identifies the resume-point, and a subsequent label. The epilogue contains a conditional return statement.

Jump table and call wrappers constitute the "release-and-resume instrumentation".

Transformation 1 (*Release-and-Resume Instrumentation*) *Every function on a call path to a potential checkpoint location is subject to the following code transformation:*

1. *Introduce a new local variable (`callid`) to store the resume-point.*
2. *Insert a jump table before the first instruction of the function with entries corresponding to each of those function calls within this function that lead to a potential checkpoint location, including potential checkpoint locations.*
3. *Insert call wrappers around all those function calls within the function that lead to a potential checkpoint location, including potential checkpoint locations.*

We illustrate the release-and-resume instrumentation by means of an example. Function `foo` below calls function `bar` and contains a potential checkpoint location, specified by a call to library function `checkpoint`. Function `bar` is assumed to contain another potential checkpoint location.

```
void foo(void)
{
      :
   bar();
      :
   checkpoint();  /* potential checkpoint location */
      :
}
```

Figure 5 contains a simplified version of the release-and-resume instrumentation, which emphasizes the control flow redirection. The jump table is inserted before the first statement of `foo`. Both function calls `bar()` and `checkpoint()` are instrumented with a call wrapper.

```
void foo(void)
{
   unsigned long callid;

   if ( restoring ) {          /* jump table      */
     switch(callid) {
       case(0): goto L_call0;
       case(1): goto L_call1;
     }
   }
       :
   callid = 0;                 /* prologue       */
 L_call0:                      /* resume-point 0 */
   bar();                      /* call 0         */
   if ( checkpointing )        /* epilogue       */
     return;
       :
   callid = 1;                 /* prologue       */
 L_call1:                      /* resume-point 1 */
   checkpoint();               /* call 1         */
   if ( checkpointing )        /* epilogue       */
     return;
       :
}
```

Figure 5: Simplified illustration of the release-and-resume instrumentation, consisting of a *jump table* at the function entry and *call wrappers* around function calls.

The following execution scenario illustrates the control-flow redirection: During *normal execution*, flags `checkpointing` and `restoring` are false. Upon entering `foo`, the jump table is skipped. Assume that normal execution arrives at the prologue of `checkpoint`, when a checkpointing signal is received. In the prologue `callid` is set to 1, which identifies the resume-point. Then, `checkpoint` is called. Within `checkpoint`, flag `checkpointing` is lit as a consequence of having received the checkpointing signal. Upon return from `checkpoint`, the epilogue conditional is true, which causes a release of the stack frame of `foo` by returning to the caller. The release of stack frames continues until the library supplied main function at the bottom of the runtime stack is reached. There, flag `checkpointing` is reset and flag `restoring` is lit. Then, the stack is restored by resuming the same function call sequence that has been active when calling function `checkpoint`. When resuming function `foo`, resume-point identifier 1 is loaded into variable `callid`, enabling the jump table to redirect control to the prologue of `checkpoint`. Function `checkpoint` resets flag `restoring` and returns. Now, back to normal execution, the epilogue conditional is false, and the statement following the epilogue conditional is executed.

In the previous discussion, we omitted how resume-point identifier 1 is assigned to `callid` when function `foo` is resumed. During checkpointing, all local variables are saved before the releasing return statement in the epilogue. Similarily, restoring local variables is integrated with the jump table. As all other local variables, `callid` is saved and restored. The instrumentation of save and restore code consists of push and pop operations on a ***shadow stack***, whereby local variables are accessed by name. A description of the save and restore code generation can be found in [15]. Note that recovery involves no more than resuming the functions on the runtime stack.

Several code transformations are required prior to the release-and-resume instrumentation that are not elaborated here. Among the obvious transformations are moving initializers beneath the jump table, moving function calls on a call path to a potential checkpoint location out of expressions, introduce dummy return values for the releasing return, and moving declarations in nested blocks to the top level to unify the name space of local variables. Furthermore, optional live-variable analysis may be performed to identify those local variables for which checkpointing and recovery code must be generated.

This section briefly described the portable checkpoint compiler porch, and the method for providing a portable stack environment in detail. Applying this method to implement closure creation in Cilk when spawning a thread of control minimizes the overhead of the `spawn` operation. The resulting implementation of indolent closure creation is introduced next.

4 Indolent Closure Creation

This section describes how indolent closure creation can be integrated with Cilk. Furthermore, an obvious optimization of indolent closure creation is introduced that I call *sedative closure creation*. Indolent closure creation carries the work-first principle to an extreme. Rather than creating closures eagerly whenever a `spawn` operation is encountered, indolent closures are created *on demand*.

> **Indolent Closure Creation:** Create closures only if an unsuccessful steal attempt occurs. Closures are then created by unrolling the runtime stack and pushing closures onto the ready deque analogous to saving the state of the runtime stack on the shadow stack in porch.

The key idea of indolent closure creation is to delay the creation of closures to the occurance of an ***unsuccessful steal attempt***. A steal attempt is unsuccessful if the ready deque of the victim is empty. When the victim detects an unsuccessful steal attempt, it unrolls its runtime stack, potentially creating several closures. Subsequent steal attempts are therefore likely to be successful. Indolent closure creation reduces the overhead of closure creation from the number of spawn operations to the number of unsuccessful steals, following the philosophy of the work-first principle. Furthermore, I believe that the number of unsuccessful steals experienced by a thief is increased only by a constant factor, if the work of a thread is limited by an upper bound. In this case, the properties of randomized work-stealing continue to hold under indolent closure creation.

The implementation of indolent closure creation combines the following key techniques from Cilk-5 and porch:

1. We introduce *potential unroll locations* which correspond to *potential checkpoint locations* in porch. Closures can only be generated upon visiting a potential unroll location during execution.

   As in porch, potential unroll locations must be specified in a program to allow for live variable analysis and code generation at compile time. For indolent closure creation, potential unroll locations are generated by the transformation of the `spawn` operation. They can occur either before or after the spawned procedure call. The generated code for unrolling the runtime stack is guarded by a check as to whether an unsuccessful steal attempt occured.

2. We employ *cloning of procedures* during source-to-source compilation to minimize the scheduling overhead in the fast clone according to the work-first principle.

   Cloning can be viewed as an optimization of the *release-and-resume transformation* of porch. In short, the fast clone needs to contain the save code only, that is the function call wrappers according to Definition 2. The slow clone would be used to rebuild the runtime stack, and must therefore contain the jump table (Definition 1) as well.

3. The *shadow stack* used by porch to save all live variables of the runtime stack corresponds to the *ready deque* in Cilk.

   From Cilk's point of view, the ready deque can be maintained as a (cactus) stack. The slow clone of porch can set up the linkage across the closures on the (shadow) stack.

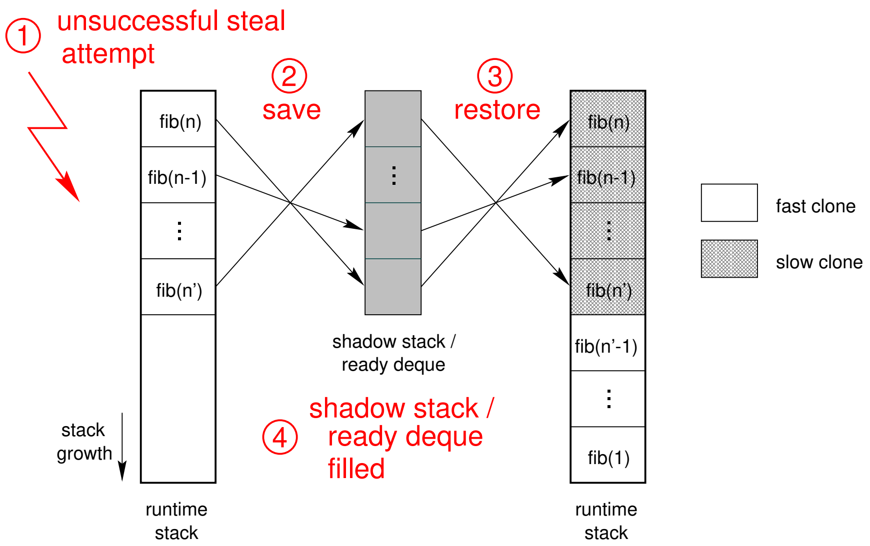

Figure 6: Indolent closure creation by the victim.

Figure 6 shows the process of indolent closure creation on the victim processor after a steal request occured. Note that Figure 6 shows the runtime stack and shadow stack of the victim only. In contrast to

Figure 3, the thief is not shown, because its behavior does not deviate from the description in Section 2. We assume that the Fibonacci computation has proceeded to execute `fib(n')`, when a steal (1) is attempted by the thief. Upon detection of the steal request, the victim saves its runtime stack (2) by releasing all functions on the runtime stack and pushing the live variables of each activation frame onto the shadow stack (ready deque), accessing them by their name. Note that copying the live variables could be accompanied by an optional data representation conversion to facilitate transferring the closures across binary incompatible machines [18]. When all activation frames are released, the runtime stack of the victim is empty. It is rebuilt during the subsequent restore phase (3). The slow clone is executed to copy the live variables from the shadow stack back into the corresponding runtime-stack activation frames. After the stack is rebuilt, execution of the Fibonacci computation resumes with the fast clone of `fib(n'-1)`. The ready deque is now filled with a number of closures available for stealing (4). Consequently, several subsequent steal attempts will succeed with high probablity.

The fast clone of `fib`, instrumented for indolent closure creation, is shown in Figure 7. It is instrumented with a local variable `entry`, which is the portable program counter, analogous to variable `callid` in Figure 5. Only the first `spawn` is transformed into a prologue (lines 8–13), the function call itself (line 14), and an epilogue (lines 15–19).

```
1  int fib(int n)
2  {
3     int x, y;
4     int entry;
5     if (n < 2)
6        return n;
7     else {
8        if ( StealAttempted ) {    /* prologue */
9           cfmode = SAVE;
10          entry = -1;
11          save(n,entry);
12          return 0;
13       }
14       x = fib(n-1);
15       if (cfmode == SAVE) {    /* epilogue */
16          entry = 1;
17          save(n,entry);
18          return 0;
19       }
20       y = spawn fib(n-2);      /* not transformed */
21       /* sync is nop */ ;
22       return x+y;
23    }
24 }
```

Figure 7: Fast clone of the *indolent* version of `fib`. Compiler-generated code is shown in grey.

The prologue checks a global flag `StealAttempted` to determine whether an unsuccessful steal attempt occured. If so, another global control-flow flag `cfmode` is set to mode `SAVE` (line 9). For processors that provide global registers to hold the control-flow flag, this assignment will be inexpensive. The program counter `entry` is assigned (line 10) to mark the resume-point during the restore phase. Then, the live variables `n` and `entry` are pushed onto the shadow stack (line 11), and function `fib` is released (line 12), having initiated the save phase.

The epilogue is almost the same as porch's epilogue. To save the assigment of the resume-point in the prologue, cf. Figure 5, it is moved into the guarded code of the epilogue (line 16). This optimization does not change the semantics of the release-and-resume instrumentation.

Note that during normal execution only the two `if`-conditions of the prologue and the epilogue must be computed. Since the branch-prediction logic of most modern processors predict these conditions correctly, the overhead of these computations should be substantially lower than the additional work done by the fast clone of Cilk, shown in Figure 4.

Optimization: Sedative Closure Creation

The experienced programmer may have noticed that the separation of the prologue and the epilogue in Figure 7 might be superfluous. In order to reduce the overhead of the call wrappers even further, the prologue and epilogue may be merged in order to compute only a single `if`-condition rather than two. In fact, it is possible to move the prologue code into the epilogue. The epilogue must remain, because it is instrumental for the save phase when the spawned child is released.

The rearranged code of the fast clone in Figure 7 is shown in Figure 8. I call the resulting scheme of closure creation ***sedative closure creation*** for the reason discussed in the following section. The semantics of the global control-flow flag `cfmode` is extended to include the state `STEALATTEMPTED`, indicating that an unsuccessful steal has been attempted. In the indolent version, `StealAttempted` is a separate flag. As a result, for every spawn no more than the evaluation of the epilogue condition is required to check whether the computation is proceeding normally, or not.

```
1  int fib(int n)
2  {
3     int x, y;
4     int entry;
5     if (n < 2)
6        return n;
7     else {
8        x = fib(n-1);
9        if (cfmode != NORMAL_EXECUTION) {    /* epilogue */
10          if (cfmode == SAVE) {
11             entry = 1;
12             save(n,entry);
13          }
14          else if (cfmode == STEALATTEMPTED) {
15             cfmode = SAVE;
16             entry = -1;
17             save(x,n,entry);
18          }
19          return 0;
20       }
21       y = spawn fib(n-2);      /* not transformed */
22       /* sync is nop */ ;
23       return x+y;
24    }
25 }
```

Figure 8: Fast clone of the *sedative* version of `fib`. Compiler-generated code is shown in grey.

The epilogue of the sedative version occupies lines 9–20 of Figure 8. No prologue is generated in this version of the code. Line 9 of Figure 8 contains the check of the control-flow flag. The guarded epilogue is entered only if the computation does not proceed normally, in which case two modes of execution are distinguished. Either the save phase is *initiated* (`cfmode == STEALATTEMPTED`), or the computation is already engaged in executing the save phase (`cfmode == SAVE`). The actions in each of the cases are the same than in the indolent version.

Comparison of Indolent and Sedative Closure Creation

The sedative optimization of indolent closure creation implies a subtle difference concerning the provision of parallelism. The potential unroll location of the indolent version resides in the prologue of the `spawn`. In the sedative version it is embedded in the epilogue. It is not clear at the time of this writing, how this difference affects the performance of the randomized work-stealing scheduler.

Whereas the indolent version visits potential unroll locations *before* calling a spawned procedure, the sedative version visits potential unroll locations *after* spawning a procedure. Thus, the sedative version executes the spawned procedure before it may generate parallelism by creating closures. This may lead to starvation of the thiefs. On the other hand, if a spawned procedure is fine-grained, the sedative version prevents costly steal operations, because it is likely to unroll the runtime stack and generate closures only when the runtime stack is loaded with a larger number of activation frames. Hence the name *sedative closure creation*. This optimization is likely to prevent unnecessary transfers of closures, and avoid parallelism to be become counterproductive, if a steal operation were more expensive than executing a thread locally.

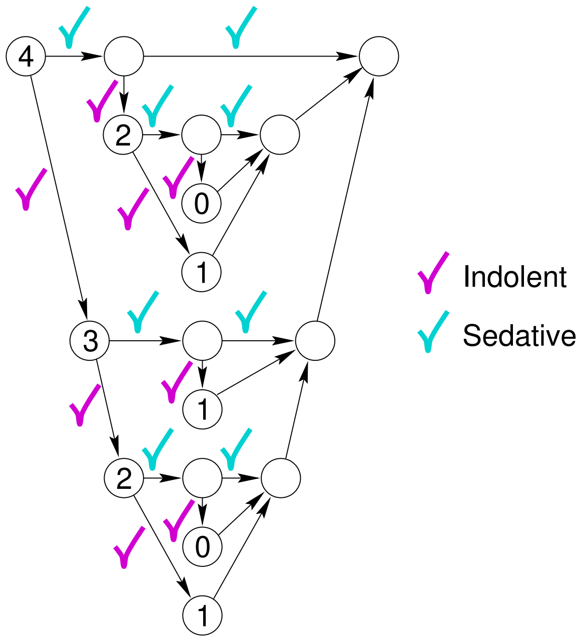

Figure 9: Comparison of indolent and sedative strategies for `fib(4)`. Checks for unsuccessful steal attempts are marked for both strategies.

Figure 9 illustrates the difference between the indolent and sedative versions by means of the DAG of Fibonacci computation `fib(4)`. In this DAG, spawn edges point towards the bottom of the page, and nodes with in-degree > 1 are sync nodes. Those egdes of the DAG, where potential unroll locactions reside, are checked dark-grey for indolent closure creation and light-grey for sedative closure creation. The checks can be interpreted as follows. Node 4 (short for `fib(4)`), for example, spawns node 3. Before starting execution of node 3, a check for an unsuccessful steal attempt is performed in the indolent version; hence the spawn edge between nodes 4 and 3 is attributed with a dark-grey check. After returning from node 3 and before spawning node 2, a check would be performed in the sedative version. The edge between node 4 and the subsequent spawn-node is therefore marked with a light-grey check.

Implications on Scheduling

The existance of potential unroll locations discretizes the runtime of a computation into an irregular one-dimensional grid. Grid points correspond to potential unroll locations, as do potential checkpoint locations in porch. Only when a potential unroll location is visited can closures be generated. In order to avoid starvation of the thiefs, the distance between grid points must be appropriately small. The distribution of the grid points on the time axis is different for the indolent version and the sedative version of a multithreaded computation.

As an example, consider the execution of the indolent version of the Fibonacci computation in Figure 9. We assume that the DAG represents a computation where all numbered nodes correspond to some substantial amount of work, and the computation is executed by a node that becomes a victim of steal attempts. The computation of the victim proceeds from node 4 downwards via nodes 3 and 2 to node 1. The execution time of each of these nodes determines the distance of the grid nodes where a check for a steal attempt is performed. Obviously, all thiefs starve until the victim visits a potential unroll location and performs the check. For the sedative version, the first check is performed only after node 1 at the very bottom of the DAG returns to parent node 2. Thus, the distance of the first grid point from the start of the computation would be the sum of the execution times of nodes 4, 3, 2, and 1 in the sedative version.

Starvation is minimized, if the execution times of the individual nodes is small. For this reason, indolent closure creation as well as its optimized sedative version is limited to fine-grained multithreaded computations. Although not done yet, I assume that the introduction of an upper bound on the execution time of the nodes will increase the critical-path length by a constant factor, due to starving thiefs. Under this condition, the performance guarantees of the work-stealing scheduler should be effectively unchanged, and the performance of a Cilk computation should be unaffected. The theoretical aspects of scheduling as well as experimental validation are subject to future work.

DAG-Consistent Shared Memory

When implementing Cilk with DAG-consistent shared memory [3], for example on a symmetric multiprocessor (SMP), the question arises whether indolent closure creation can be implemented without restricting the full range of language features supported by Cilk. In particular, pointers passed by reference appear to be problematic. They do not pose a problem, however, as briefly discussed in the following.

Cilk-5 operates on a shared address space. In particular, the cactus stack of closures is shared among processors. In order to resume execution of a stolen frame, a thief does not copy the live variables from the closure into the activation frame. Instead, the slow clone of a Cilk procedure contains redirections of all references and variable accesses to their corresponding address in the shared closure. If the slow clone of the indolent version uses the same implementation for sharing of values via the closure, handling pointers is even simpler than in porch.

Pointers passed by reference are handled almost as in porch [18, Section 5] when unrolling and restoring the runtime stack. During the save and restore phases, pointers are transformed into offsets within the shadow stack. On a shared memory machine, the offset would not be an integer, but the pointer to its target's shadow address. Compared to porch's pointer resolution algorithm, the computation of the offset can be avoided. If a slow clone passes a pointer to the fast clone of a child, it will pass the pointer to an address in the shared closure, rather than to the private copy on its runtime stack. A thief executes the slow clone after a successful steal attempt. It accesses a stolen closure just as the victim did during the restore phase. Thus, restarting a stolen computation on a thief involves no more than the restore phase of indolent closure creation.

The preceding section described the source-to-source translations which enable indolent creation of closures on demand. We discussed an optimization of indolent closure creation, sedative closure creation, that delays creation of closures even further than indolent closure creation to the point during execution where it is more likely that parallism does not become counterproductive, and a larger number of closures is prepared when unrolling the stack. The following section provides experimental evidence for the validity of the proposed implementation techniques.

5 Experimental Results

This section presents preliminary results of hand-coded versions of the fast clone of the Fibonacci program. Neither stealing nor the execution of the slow clone are included. Instead, only sequential execution time and, thus, the overhead when executing the fast clone—the common case—is measured. Both versions with indolent closure creation and with sedative closure creation have been compiled with several compilers and have been run on various processor architectures. The execution times reported are the minimum times of at least four repeated measurements.

Indolent Fibonacci Computation

Table 1 presents serial execution times for two versions of `fib(36)`, the sequential execution time T_S of the plain C program, and execution time T_1 of the indolent version of `fib` on one processor. The overhead of the indolent version is shown with respect to the C version. Also included are the corresponding overheads of the Cilk-5 implementation on one processor from [7, Figure 7]. For the Pentium, the measurement of the Cilk overhead has been performed on a 200 MHz processor, whereas those of the fast clone of the indolent version were run on a 120 MHz processor.

Machine/Compiler		T_S $[s]$	T_1 $[s]$	Overhead $[\%]$	Cilk-5 Overhead $[\%]$
	gcc -O2	1.3	2.7	111	
Alpha	gcc -O3	1.3	2.7	111	520
477 MHz	cc -O	1.4	2.7	94	
	cc -O5	1.3	2.8	113	
MIPS	gcc -O2	7.2	8.8	23	
195 MHz	gcc -O3	7.2	8.8	22	120
Pentium	gcc -O2	4.8	7.6	59	(200 MHz)
120 MHz	gcc -O3	4.8	7.6	59	330
PowerPC	gcc -O2	5.7	7.8	35	
166 MHz	gcc -O3	5.9	7.8	33	
UltraSparc-I	gcc -O2	6.5	8.3	28	
143 MHz	gcc -O3	5.9	9.4	60	
	gcc -O2	5.3	7.3	39	
UltraSparc-I	gcc -O3	5.0	6.5	31	230
167 MHz	cc -O	4.1	6.0	46	
	cc -xO5	4.3	7.4	71	

Table 1: Execution times and overheads of the fast clone of the *indolent* version of `fib(36)` for various machine and compiler combinations.

The overheads of the indolent version are below 113 %. Thus, a parallel `spawn` operation is roughly at most a factor of two slower than a sequential function call. For all processors except the Alpha, the overhead is even below 70 %. The differences caused by the choice of the compiler and the optimization levels are significant only for the UltraSparc. It is noteworthy that on the 143 MHz UltaSparc, switching the optimization level of gcc from O2 to O3 causes the plain C version to become faster, whereas the fast clone becomes slower.

Sedative Fibonacci Computation

Table 2 presents serial execution times T_1 of the sedative version of `fib(36)` on one processor. The sequential execution times T_S of the plain C program are not always the same as in Table 1 due to the variance of the measurements. The overhead of the sedative version is shown with respect to the C version. The overheads of the indolent version, shown in Table 1, are included as well to simplify comparison.

Machine/Compiler		T_S $[s]$	T_1 $[s]$	Indolent Threads [%]	Sedative Threads [%]
	gcc -O2	1.3	2.0	111	57
Alpha	gcc -O3	1.3	2.0	111	57
477 MHz	cc -O	1.4	2.1	94	53
	cc -O5	1.3	1.9	113	48
MIPS	gcc -O2	7.7	7.5	63	-3.6
195 MHz	gcc -O3	7.7	7.4	63	-4.2
Pentium	gcc -O2	4.8	5.9	59	24
120 MHz	gcc -O3	4.8	5.9	59	24
PowerPC	gcc -O2	5.7	6.3	35	10
166 MHz	gcc -O3	5.9	6.3	33	6.2
UltraSparc-I	gcc -O2	5.9	8.9	28	52
143 MHz	gcc -O3	5.7	7.3	60	29
	gcc -O2	5.1	7.8	39	53
UltraSparc-I	gcc -O3	4.7	6.3	31	33
167 MHz	cc -O	4.7	6.9	46	45
	cc -xO5	4.7	6.4	71	37

Table 2: Execution times and overheads of the fast clone of the *sedative* version of `fib(36)` for various machine and compiler combinations.

The overhead of a parallel spawn in the sedative version is below 60 % in all experiments. The expected reduction of the overhead of the sedative version with respect to the indolent version is a factor of two, because the sedative version executes one conditional branch, and the indolent version two. This is roughly the case for all architectures except the MIPS and the UltraSparc processors. For the latter, the effect of the sedative optimization is hardly recognizable. On the MIPS processor, the parallel spawn is even faster than the sequential function call.

6 Related Work

The implementation of Cilk-5 [7] has been motivating the work on indolent closure creation. Currently, I view indolent closure creation as an optimization for implementations of Cilk. It would be interesting to investigate whether indolent closure creation could be applied to parallel functional languages such as Mul-T [11], Id [13], or the parallel Haskell dialect pH [14]. A variety of approaches to low-overhead implementations of multithreaded languages have been studied on commodity computers. I discuss only a subset of them and refer the reader to the papers cited therein. Both papers [8] and [9] discuss a large body of related work.

Lazy Threads [8] extend the work on the Threaded Abstract Machine (TAM) [5], a compilation target for parallel nonstrict functional languages. Lazy Threads are based on compiler support which implements customized memory management of activation frames with so-called *stacklets*. This customization enables a more general handling of frames in the context of non-strict languages than required by the semantics of a sequential call. The paper [8] introduces a classification of different closure representations that allows the compiler to select the appropriate representation for particular instantiations. The Fibonacci computation is used in [5, 8] to illustrate the proposed mechanisms.

The Illinois Concert system [9] employs a customized compiler to reduce the cost of thread management by means of a *hybrid stack-heap execution mechanism*. Similar to Cilk, the compiler generates two clones for each thread body, one of which executes off a stack-allocated activation frame, and the other from a heap-allocated context. A thread executes optimistically on its caller's stack, and is converted lazily into a heap-allocated thread only when necessary, for example when being blocked in a communication primitive. The compiler is used to determine whether a parallel call can be replaced by a sequential function call. This is the case, if the compiler can ascertain that a thread will not block.

7 Conclusions

I propose an implementation for generating parallelism in multithreaded languages that minimizes the cost of a parallel spawn to be comparable to the cost of a sequential function call. Key to reducing the cost of a parallel spawn is the indolent creation of closures—representations of threads, ready to be executed—on demand, that is only when an unsuccessful steal attempt occurs. Sedative closure creation is an optimization of indolent closure creation where potential closure creation is delayed until after a spawned procedure has been executed.

Open questions involve the theory for scheduling of threads created indolently or sedatively. I believe that the introduction of an upper bound on thread granularity can be used to show that starvation of thiefs can be bound, and the associated increase of the critical-path length be limited within a constant factor. An upper bound on thread granularity implies that, in practice, indolent closure creation would preserve the properties of randomized work-stealing for fine-grained multithreaded computations. I believe that this limitation is reasonable, because it seems to be more likely that, eventually, efficient compilers can be built that coalesce fine-grained threads automatically rather than coarse-grained threads.

The experimental results presented in this memo demonstrate that a parallel spawn can be as cheap as a sequential function call on modern processors. It is not clear, however, which implications indolent closure creation has on the performance of a fine-grained multithreaded Cilk computation per-se. Without a complete implementation, it is hard to predict whether fine-grained multithreading can be implemented efficiently and portably on parallel architectures. In particular, heterogeneous environments with their inherent problems concerning performance predictability continue to pose numerous chances and challenges.

Acknowledgements

My sincere gratitude to Charles Leiserson for preserving my freedom of thought over the last two years in a truly creative manner. I am delighted to thank Matteo Frigo for helpful discussions about the current implementation of Cilk and his constructive ideas when I flushed my ideas on indolent closure creation. Thanks also to Harald Prokop for his suggestions on an early draft of this memo.

References

[1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *4th ACM International Conference on Supercomputing*, pages 1–6, Amsterdam, The Netherlands, June 1990.

[2] Andrew W. Appel. Garbage Collection Can Be Faster than Stack Allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[3] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-Consistent Distributed Shared Memory. In *10th International Parallel Processing Symposium*, pages 132–141, Honolulu, Hawaii, April 1996.

[4] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *35th Annual Symposium on Foundations of Computer Science*, pages 356–368, Santa Fe, New Mexico, November 1994.

[5] David E. Culler, Seth C. Goldstein, Klaus E. Schauser, and Thorsten von Eicken. TAM—A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, 18(3):347–370, July 1993.

[6] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous Multithreading: A Platform for Next-Generation Processors. *IEEE Micro*, pages 12–19, September/October 1997.

[7] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Canada, June 1998. ACM SIGPLAN.

[8] Seth C. Goldstein, Klaus E. Schauser, and David E. Culler. Lazy Threads: Impementing a Fast Parallel Call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, August 1996.

[9] Vijay Karamacheti, John Plevyak, and Andrew A. Chien. Runtime Mechanisms for Efficient Dynamic Multithreading. *Journal of Parallel and Distributed Computing*, 37(1):21–40, August 1996.

[10] James S. Miller and Guillermo J. Rozas. Garbage Collection is Fast, But a Stack is Faster. MIT Artificial Intelligence Laboratory, AI memo 1462, March 1994.

[11] Eric Mohr, David A. Kranz, and Jr. Robert H. Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[12] Joel Moses. The Funtion of FUNCTION in LISP, or Why the FUNARG Problem Should be Called the Environment Problem. MIT Artificial Intelligence Laboratory, AI memo 199, June 1970.

[13] Rishiyur S. Nikhil. A Multithreaded Implementation of Id using P-RISC Graphs. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, LNCS 768, pages 390–405, Portland, Oregon, August 1993. Springer Verlag.

[14] Rishiyur S. Nikhil, Arvind, James Hicks, Shail Aditya, Lennart Augustsson, Jan-Willem Maessen, and Yuli Zhou. pH Language Reference Manual. MIT Computation Structures Group, CSG memo 369, January 1995.

[15] Balkrishna Ramkumar and Volker Strumpen. Portable Checkpointing for Heterogeneous Architectures. In *Digest of Papers—27th International Symposium on Fault-Tolerant Computing*, pages 58–67, Seattle, Washington, June 1997. IEEE Computer Society.

[16] Gurindar S. Sohi, Scott E. Breach, and T.N. Vijaykumar. Multiscalar Processors. In *22nd International Symposium on Computer Architecture*, pages 414–425, Santa Margherita Ligure, Italy, 1995.

[17] Guy L. Steele, Jr. Debunking the "Expensive Procedure Call" Myth or, Procedure Call Implementations Considered Harmful or, Lambda: The Ultimate GOTO. MIT Artificial Intelligence Laboratory, AI memo 443, October 1977.

[18] Volker Strumpen. Compiler Technology for Portable Checkpoints. submitted for publication (`http://theory.lcs.mit.edu/~strumpen/porch.ps.gz`), 1998.

[19] J.-Y. Tsai and P.-C. Yew. The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation. In *International Conference on Parallel Architectures and Compilation Techniques*, October 1996.