

IBM Research Report

A Multithreaded Processor Architecture with Implicit Granularity Adaptation

Volker Strumpfen
IBM Research Division
Austin Research Laboratory
11501 Burnet Road
Austin, TX 78758



A Multithreaded Processor Architecture with Implicit Granularity Adaptation

Volker Strumpfen
IBM Austin Research Laboratory

September 18, 2006

Abstract

We propose a multithreaded processor architecture for mapping a potentially unbounded number of software threads into a bounded number of hardware threads. Our architecture degrades a fork instruction gracefully into a function call when all hardware threads are in use, thereby adjusting thread granularity implicitly. This new architectural feature improves programmability by allowing programs to expose as much parallelism as is inherent in an application without incurring a performance penalty due to excess parallelism.

1 Introduction

Our goal is to simplify the programming of parallel applications. We propose a solution that is similar in spirit to the way in which virtual memory hides the problem of accessing different levels in a memory hierarchy behind a simple load/store interface. The performance of virtual memory systems degrades gracefully as larger amounts of data occupy the memory hierarchy. Replacement algorithms, such as least-recently-used, automate data transfers across the levels while optimizing the common case by exploiting locality. The virtual memory system is a typical application of the design philosophy of *graceful degradation*: functionality is reduced gradually in the face of faults. By interpreting the exhaustion of a fast resource as a fault and performance as a functionality, we can tailor the philosophy of graceful degradation to our needs: performance is reduced gradually in face of larger resource requirements.

In this article, we apply the design philosophy of graceful degradation, and show that it can pay to elevate the degraded, special case into the common case rather than treating it as a rare exception. We focus our attention on the design of a multithreaded processor architecture. Space-time constraints force us to limit the number of threads that we can support efficiently in hardware to a relatively small, bounded number. When exhausting these thread resources, we gracefully degrade the fork operation of a thread into a function call. However, we do not perform this degradation to salvage programs that would otherwise fail when forking a larger number of threads than the processor supports. Instead, our goal is to enable the program to specify as much parallelism as is inherent in an algorithm, because our processor handles excess parallelism efficiently. The number of hardware threads can then be determined independently of applications, so as to hide the average memory latency, for example. Our proposed processor architecture avoids performance degradation due to excess parallelism by degrading threads into functions. This *fork degradation* bounds the

bookkeeping overhead associated with parallelism by restricting the amount of parallelism adaptively.

Parallel programming languages such as Multilisp [14], Mul-T [18], and Cilk [12] offer adaptive parallelism. These languages hide the bulk of the parallel programming complexity by automating such tasks as load balancing and data transfers. Threads in these languages are different entities than traditional software threads at the user-level or as provided by the operating system, see [23, 26, 21, 11] to name just a few. The latter allocate one context per thread, and schedule threads preemptively by means of context switches. In contrast, languages like Cilk and Mul-T create and schedule threads non-preemptively on processors only if a processor idles. In addition, Cilk’s work-stealing scheduler assigns threads to processors with theoretical performance guarantees [7]. Here, we focus on executing one parallel application efficiently. The subtle difference lies in the scheduler constraints: The scheduler of an operating system or for conventional user-level threads assumes that threads can be scheduled in any order. For a parallel application, the order in which threads are scheduled may impact correctness and is crucial for performance.

In this article, we discriminate between *hardware threads* and *software threads*. Hardware threads require hardware structures for bookkeeping, and software threads are mapped into hardware threads to be executed within its context, rather than allocating a thread context in software. We say that a hardware thread *shepherds* the execution of a software thread. Our proposal is based on three observations. (1) We can lump multiple fine-grained software threads into one coarse-grained hardware thread during execution. (2) Since hardware threads require hardware structures for bookkeeping, we are interested in bounding the number of hardware threads so that fast circuits can be employed for implementing thread management operations. (3) We can map software threads into hardware threads without penalizing the creation of parallelism, neither in space nor time. From this perspective, our proposal may be viewed pictorially as an *infinite thread architecture* that enables programmers or compilers to focus on exposing the parallelism inherent in the problem, rather than on tuning performance for a particular machine. In particular, we avoid the space and time overheads associated with software-allocated thread contexts and the switching between these contexts.

We denote as *thread granularity* the number of executed instructions of a thread. A program with coarse-grained threads implies a relatively small number of threads, which enjoy a relatively low bookkeeping overhead in both memory requirements and execution time. In particular for irregular applications, however, large grain sizes can cause poor load balancing. To the contrary, small grain sizes are usually associated with a large number of threads which can improve load balancing at the expense of larger bookkeeping overheads. Ideally, we can relieve the programmer from considering the intricate granularity trade-offs altogether.

Our experience with Cilk shows that it is relatively easy for a program to create *excess parallelism* in form of threads [12, Section 6]. Research on mapping applications to dataflow architectures as well as on constructing dependency graphs in the compiler arena [3, 22, 32] provide further evidence for the validity of this assumption. In addition, we assume that programmed units of parallelism are encapsulated within functions, as supported by Cilk. Careful design of a function requires choosing the minimal thread granularity to be coarse enough to amortize the function call overhead. As a side effect, we avoid the excessive space and time penalties of extremely fine-grained instruction-level parallelism. While functions are natural units of parallelism in most programming languages, some languages expose similar opportunities such as expressions in Scheme [1], which are exploited by Multilisp and Mul-T.

In short, we tackle the problem of mapping a potentially large number of software threads automatically and efficiently into a limited number of hardware threads. A similar problem has been studied before in the context of Mul-T [20] and Cilk [7]. The mapping proposed as part of these languages is a software solution of user-level threads into kernel threads or processes. Here, we propose a microarchitectural solution for a multithreaded processor that offers a different perspective and has several advantages in its own right: (1) thread creation and termination does not incur any performance penalty, (2) context switching comes for free, (3) we implement granularity adaptation by degrading a fork into a function call with a minor performance penalty of executing one `nop`, (4) we integrate thread management with memory latency hiding in the thread scheduler. As a result, our architecture invalidates the pretense that “*lazy future calls are unlikely ever to be as cheap as the cheapest implementation of normal calls*” in [20, Section 7, p. 279].

The remainder is structured as follows. In Section 2, we introduce *fork degradation* abstractly within a *thread model*. In Section 3, we discuss the general problem of mapping software threads into hardware threads. Section 4 is dedicated to the implementation of our thread model, and Section 5 describes one possible instantiation of a multithreaded processor based on a simple RISC architecture. We position our architecture in the context of previous work in Section 6.

2 The Thread Model

In the following, we discuss our thread model from the perspective of a multithreaded architecture. Our thread model introduces a new feature called *fork degradation*. Thread models have been discussed previously in [7, 9, 16], for example. We are interested in threads as known from Cilk and Mul-T [12, 18]. In contrast to traditional preemptive schedulers for user-level or kernel threads, these languages rely on schedulers tailored to executing a parallel program efficiently by careful choice of the strategy for selecting threads for execution.

A *hardware thread* represents the hardware resources needed to shepherd the execution of a software thread. *Software threads* are created and terminated by means of the instructions:¹

`fork <label>` creates a software thread that is mapped into a hardware thread, which then shepherds the execution of a code block beginning at instruction address `label`,

`join lr` synchronizes the forking and the forked thread. Register `lr` is the link register; its use is explained below.

We illustrate our thread model and the semantics of the `fork` and `join` instructions by means of the example in Figure 1. Figure 1(a) shows a code fragment consisting of two functions `foo` and `bar`. Function `foo` contains code blocks A, B and C. By definition, a *code block* shall not contain any `fork` or `join` instructions. Before code block B, `foo` forks function `bar`, so that code block D may execute concurrently with code block B. The control flow of functions `foo` and `bar` synchronizes by means of the respective `join` statements, which enforces that code block C is executed only after executing both `join` statements behind (in textual order) code blocks B and D. The code fragment

¹Conway [8] introduced the `fork` and `join` pair of instructions, see also [10]. As a tribute, we use the same instruction names, although we use the instructions with the semantics of Dijkstra’s structured `cobegin` and `coend` commands and Hoare’s concurrency operator `||` [16]. Originally, Conway [8] introduced the `join` instruction with a counter argument. The counter must be initialized with the expected number of threads to join, and is decremented atomically upon each `join` until it reaches value 0. The thread executing the `join` when the counter reaches value 0 continues execution.

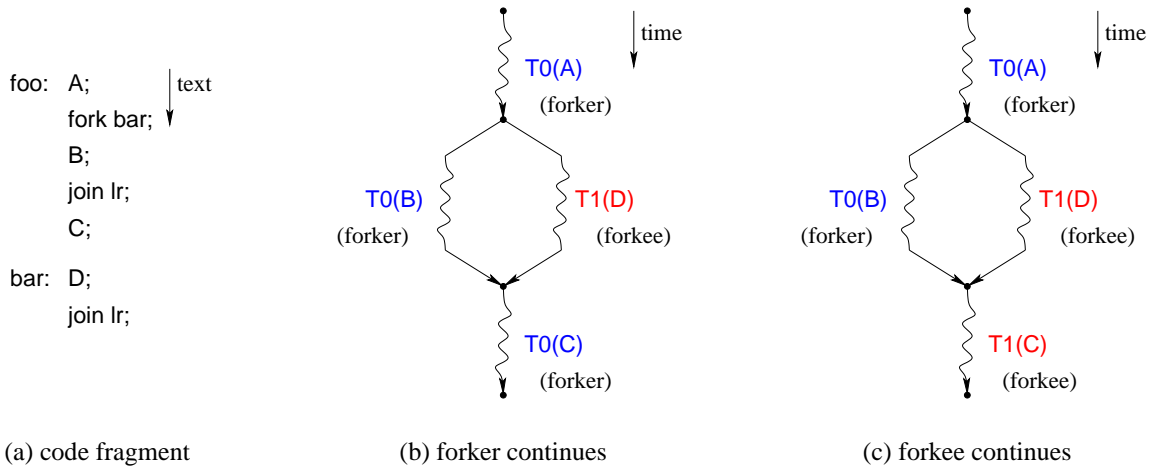


Figure 1: Code fragment (a) can be executed by two threads (T0, T1) in several ways. Illustrated are two options where the forker continues execution after the join point (b) or the forkee (c).

specifies two software threads, one associated with function `foo`, and the second with function `bar`. We denote a software thread executing a `fork` instruction a *forker thread*, and the associated forked thread the *forkee thread*.²

The diagrams in Figure 1(b) and (c) illustrate two potential assignments of the individual code blocks of the software threads to hardware threads T0 and T1. Vertices in these diagrams represent fork and join instructions, and the wiggly lines represent code blocks. Hardware thread T0 is a hardware forker thread, because it executes statement `fork bar`; of the software forker thread associated with function `foo`. In Figure 1, thread T0 shepherds the execution of initial code block A. When the `fork` instruction executes, forker T0 creates a new software thread, the forkee thread, which is assigned to hardware thread T1 for shepherding the execution of code block D.

In Figure 1(b) and (c), hardware thread T0 continues execution with the instructions after the `fork` statement, that is with code block B. Alternatively, we could have chosen the opposite assignment, where the hardware forker shepherds the software forkee, and a new hardware thread continues execution of the software forker. However, we prefer the option illustrated in Figure 1, because instantiating a new hardware thread to execute the software forker would include copying the state of the runtime stack. Our proposal avoids this potential source of overhead, because it enables us to maintain one thread context per hardware thread rather than per software thread, and is, therefore, better suited to support fine-grained software threads.

Hardware threads T0 and T1 exist concurrently, and execution of their associated code blocks shall proceed in an interleaved fashion on our multithreaded processor. Both threads synchronize by means of the `join` instruction. Execution resumes only after both threads have reached the corresponding `join` instructions. In principle, this leaves us with four options for choosing a thread mapping to continue execution after the synchronization point: (1) terminate both hardware threads, and pick a new hardware thread to continue execution, (2) the hardware thread shepherding the forker always continues, see Figure 1(b), (3) the hardware thread shepherding the forkee always continues execution after the synchronization point, see Figure 1(c), or (4) one of the

²Our naming of *forker* and *forkee* borrows from the naming of *caller* and *callee* of function calls to emphasize the close semantic relationship.

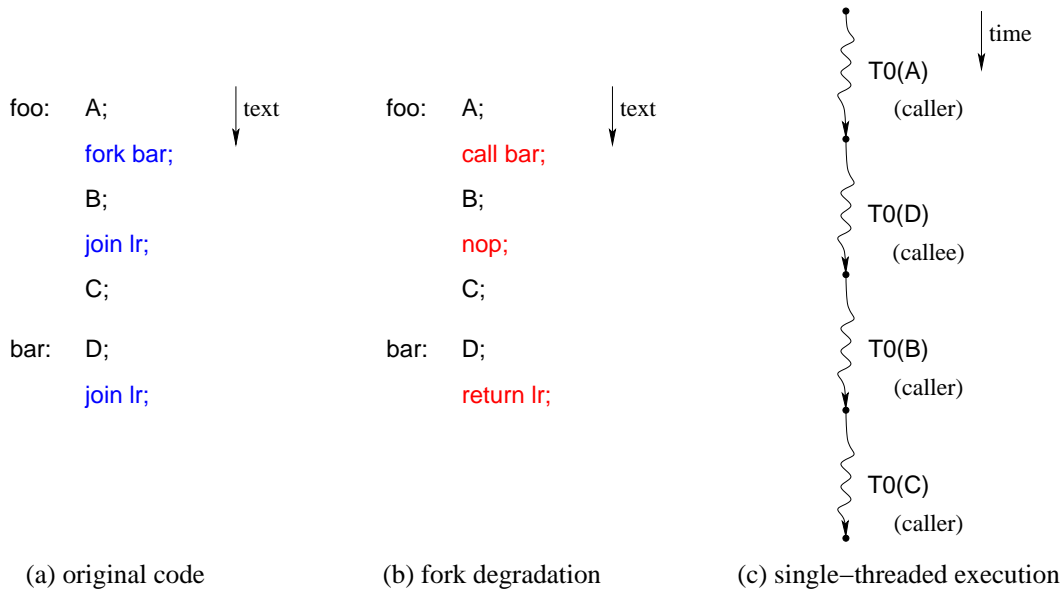


Figure 2: Interpretation of `fork` and `join` instructions (a) as a function call (b) in a single-threaded execution (c).

hardware threads, picked by some criterion at runtime, continues execution. The original `fork/join` scheme proposed by Conway [8] corresponds to option four, where the last thread reaching its `join` instruction in time continues to shepherd execution. Many multithreaded architectures, such as HEP [24], and thread models including TAM [9] follow this proposal as well. The advantage is that the first thread reaching its `join` instruction may terminate and be reused immediately without blocking any hardware thread resources.

To facilitate an efficient implementation of the hardware structures for thread management, we pick the second option:

[Forker-Continues Invariant] After synchronizing a forker and its corresponding forkee, the hardware thread shepherding the forker thread continues execution with the instruction following its `join` statement.

The primary advantage of the forker-continues invariant is that it matches the single-threaded execution scenario, which enables us to degrade a `fork` seamlessly into a function call in case when all hardware threads are assigned already. Figure 2 illustrates the single-threaded execution of the code fragment of Figure 1. We now introduce our enabling mechanism, *fork degradation*, as a mapping between a multithreaded and a single-threaded execution. Fork degradation is an isomorphism mapping forker to caller, forkee to callee, and interpreting the `fork` and `join` instructions as function call, return, and a `nop`, as contrasted in Figure 2(a) and (b). With a single thread, rather than forking function `bar`, we call function `bar` by jumping to label `bar` and saving the return address in link register `lr`. The `join` instruction in function `bar` is interpreted as a return jump to the link register address. The `join` instruction in function `foo` is redundant, because no synchronization is needed in case of a single-threaded execution. Hence, we interpret the `join` instruction in function `foo` as a `nop`. This is the beauty spot of the degradation scheme, because the caller’s `join` instruction introduces a `nop` rather than reducing to nothing.

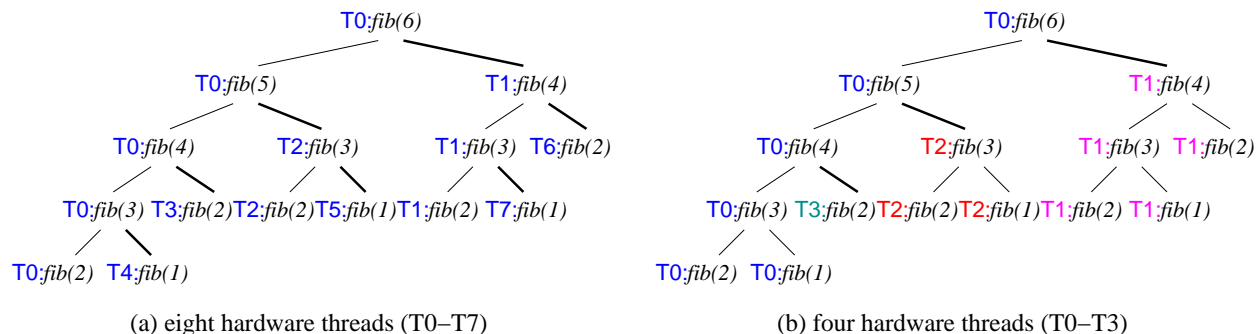


Figure 3: Multithreaded Fibonacci computation with eight software threads on eight hardware threads (left) and four hardware threads (right), where forks degrade into function calls. Function calls are represented by slim arcs and successful forks by fat arcs.

Architectural support for fork degradation implies:

1. Fork degradation increases the granularity of a hardware thread by executing an unsuccessfully forked software thread as a callee function in the hardware context of the forker thread.
2. Forking functions exposes parallelism without altering the sequential semantics of the program. We call this feature the *serial elision* in Cilk [12].
3. We need only one context, including one runtime stack, per hardware thread of the machine rather than per software thread of a parallel program.
4. The programmer or compiler may fork as many software threads as desired or inherent in an application without being aware of the limited number of hardware threads.
5. Since fork degradation incurs essentially no performance penalty, the task of specifying excess parallelism by forking a large number of software threads should be viewed as default programming style.

To substantiate these claims, we discuss the archetypical Fibonacci computation as an example. Below is the tree-recursive Scheme version [1] instrumented with a `fork` application to effect the creation of a thread.

```
(define (fib n)
  (if (< n 2)
      1
      (+ (fib (- n 1)) (fork (fib (- n 2))))))
```

Unless procedure `fib` reaches the base case (`< n 2`), we *call* `fib` with argument `(- n 1)` and *fork* a new thread to evaluate `(fib (- n 2))`.³ After both computations are complete, we add the results. The join instructions are (conveniently) implicit in the program representation. Figure 3 illustrates the tree-recursive evaluation of `(fib 6)`. We show the fork tree without closing the DAG via join vertices, because the confluent join structure is symmetric [7]. The fat arcs indicate forked threads, and the slim arcs correspond to regular function calls. The evaluation tree in Figure 3(a)

³We assume evaluation of the list of procedure arguments in reverse order, as for example implemented in MIT Scheme [1], so that the evaluation of the second argument of the addition is forked before evaluation of the first argument `(fib (- n 1))` begins.

includes seven successful forks. Accordingly, the enumeration of the software threads results in a total of eight threads. Thus, eight hardware threads would be sufficient to service each fork encountered during evaluation. For example, thread T0 shepherds (fib 6), represented by the root vertex. It forks (fib 4) and calls (fib 5). Thread T1 is assigned to shepherd (fib 4) while thread T0 continues to shepherd (fib 5).

The evaluation tree in Figure 3(b) assumes that the hardware provides only four rather than eight threads. We assume that hardware thread T0 forks thread T1 to shepherd the evaluation of (fib 4). Subsequently, thread T0 forks threads T2 and T3, at which point the four hardware threads are exhausted. Now, assume thread T1 attempts to fork (fib 2) as part of evaluating (fib 4). Since no more hardware threads are available, the fork degrades into a function call, and thread T1 shepherds procedure fib which executes as if the fork were not present in the program text at all. From a programmer’s perspective, a fork can be considered as a hint to the processor to create a thread.

The evaluation tree in Figure 3(b) emphasizes the close relationship between our hardware solution and the software solutions of Mul-T and Cilk. By replacing our hardware threads with software threads, the evaluation tree resembles the “breadth-first saturation, then depth-first” strategy of [20, Figure 2] for the tree summation. If we replace the fork keyword in procedure fib with a future keyword, we obtain a Mul-T program with an analogous evaluation tree. Furthermore, the Cilk version of fib [12, Section 2] could generate this evaluation tree as well, depending on the outcome of the random work stealing algorithm. When executed by our multithreaded architecture, the particular set of successful forks depends on instruction latencies and the hardware scheduler.

3 Thread Mapping

Using hardware threads as shepherds for software threads introduces the problem of mapping software threads into hardware threads. More succinctly, we seek a mapping for the threads of a parallel application that avoids the overheads associated with traditional preemptive scheduling. We note that the number of software threads that a program may fork is potentially unbounded. As an example, consider the program fragment in Figure 4 with a fork statement in the loop body of function bar⁴ using Conway’s join counters [8]. The forker thread shepherds function bar, and forks n forkee threads before joining. Thus, there are $n + 1$ software threads that may reach a join statement. The forker thread reaches the join statement at the end of function bar, and each of the forkees reaches the join statement of function foo. In this example, the number of software threads is unbounded since the value of variable n could be arbitrarily large. In a naive approach of mapping software threads to hardware threads, we might have to maintain an unbounded number of $n + 1$ hardware threads, which is quite objectionable for a hardware design.

Now, consider the alternative design of a machine with four hardware threads and fork degradation. We do not use a join counter. Instead, we assume that two join statements are executed for each fork, one by the forker and the other by the corresponding forkee. The code fragment in Figure 4 changes into the version shown in Figure 5. The number of software threads created by the forking loop of bar is n , as in the preceding example. However, the number of hardware threads utilized in the presence of fork degradation depends on the execution time of function foo.

Figure 6 illustrates two possible execution scenarios of the code fragment in Figure 5. The scenario in part (a) of Figure 6 assumes that function foo requires a long execution time, and the

⁴We use C syntax for this sample program, assuming that fork and join have been introduced as a new keywords.

```

int c, i, n;

void foo(int j)
{
    /* do something */
    join c;
}

void bar()
{
    c = n+1;
    for (i=0; i<n; i++)
        fork foo(i);
    join c;
}

```

Figure 4: Illustration of Conway’s fork/join scheme with a join counter, creating an unbounded number of threads.

```

int i, n;

void foo(int j)
{
    /* do something */
    join;
}

void bar()
{
    for (i=0; i<n; i++)
        fork foo(i);
    for (i=0; i<n; i++)
        join;
}

```

Figure 5: Code fragment for illustrating implicit granularity adaptation. No join counter is used.

scenario in part (b) assumes a relatively short execution time. In Figure 6(a), hardware thread T0 executes function `bar`, and begins forking `foo(i)` for $i = 0, 1, 2, 3, \dots$. Initially unused, hardware threads T1, T2, and T3 shepherd the software threads associated with iterations $i \in \{0, 1, 2\}$, respectively. Thus, all four hardware threads of the processor are busy when the next fork of iteration $i = 3$ occurs. This fork fails, and thread T0 executes `foo(3)` as a regular function call, that is the fork instruction degrades into a function call, and the software thread forked for `foo(3)` is mapped to hardware thread T0. Note that all hardware threads are utilized in this scenario, maximizing chances for effective memory latency hiding.

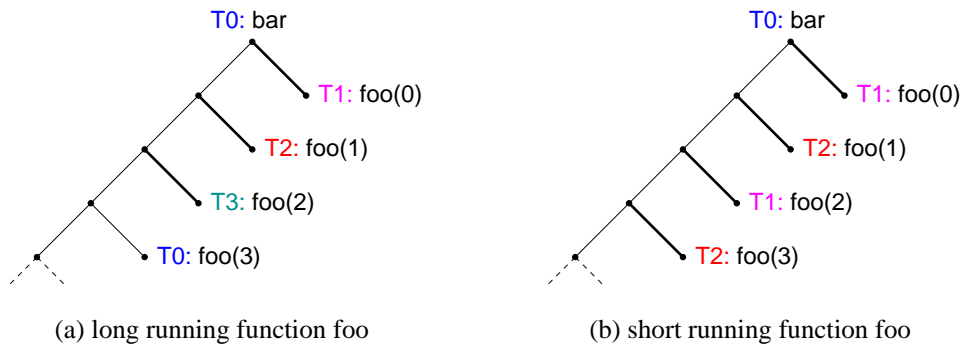


Figure 6: Example of threaded execution with four hardware threads (T0–T3), a long and a short running function `foo`.

The example in Figure 6(b) illustrates the contrasting scenario. The execution time of function `foo` shall be so short that thread T0 can fork only two threads `foo(0)` and `foo(1)`, before `foo(0)` terminates. Thus, hardware thread T1, which shepherds `foo(0)`, joins before forker T0 has been

able to fork `foo(2)`. If we require that both the forker and forkee hardware threads must join before we can reassign the hardware forkee thread, the schedule of this scenario would resemble that of Figure 6(a). That schedule is likely to be inefficient since thread T0 would shepherd $n - 3$ of n executions of function `foo` while threads T1 to T3 would be blocked waiting for the synchronization by T0. Fortunately, we can improve hardware thread utilization, if our mapping of software into hardware threads supports reusing the hardware forkee thread before the forker reaches the synchronizing join. In the example of Figure 6(b), we reuse thread T1 to shepherd `foo(2)`, thread T2 to shepherd `foo(3)`, and so on. Our proposed architecture supports reuse of forkee threads as implied in Figure 6(b).

We point out that reuse of hardware forkee threads does not provide a guarantee against blocking hardware threads. It is possible to devise programs with a fork structure that is wasteful in terms of hardware thread utilization. Figure 7 shows an example with $n + 1$ software threads. Function `foo` forks itself recursively, performs some computation, and joins with its forkee. In this scenario, the `join` statement of the forker will block the shepherding hardware thread. Due to our forker-continues invariant, we cannot reuse the hardware forker thread without saving its state, preventing its reuse. In contrast, we can reuse the forkee, because it can terminate without waiting for the forker. In the example of Figure 7, eventually, all but one hardware thread will be blocked, and the remaining active hardware thread (T3 in Figure 7) will execute the program sequentially. This example highlights the asymmetry caused by the forker-continues invariant. We may reuse forkee threads, as demonstrated in Figure 6(b), but cannot reuse forker threads without a significant performance penalty.

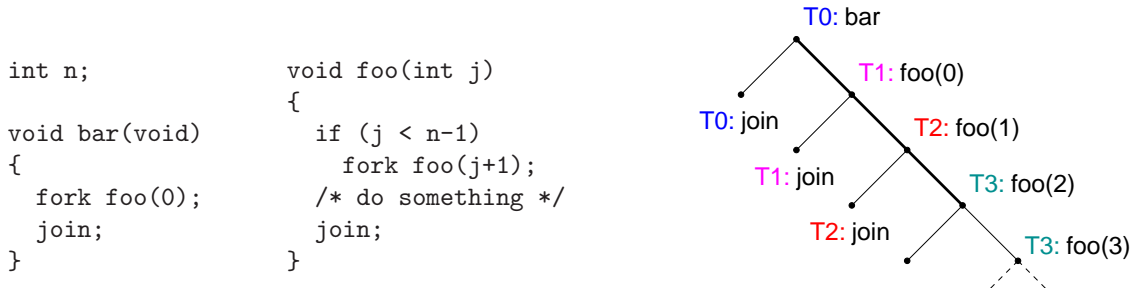


Figure 7: Example of an inadequately structured threaded program.

We can salvage this situation in one of four ways: (1) We may declare programs such as the one in Figure 7 as unreasonable and blame the programmer. (2) We may increase the number of hardware threads to ameliorate the lack of memory latency hiding due to blocked threads. This brute-force solution does not solve the pathological example of Figure 7, however. (3) We might implement a *thread switch* in software to save the state of a blocked hardware thread in dynamically allocated memory, and facilitate reuse of that thread. (4) We could devise hardware support for promoting a failed fork into a successful fork. This would be the inverse operation of our proposed method of graceful degradation of a fork into a function call. The latter option may be the most desirable, yet requires an appropriate language model, such as Multilisp [14], to be implemented with reasonably low complexity. We will not discuss these options any further, but focus on the idea of fork degradation itself.

4 Microwidgets for Thread Management

Our multithreaded processor centers around microarchitectural structures for managing hardware threads efficiently. In particular, we introduce a hardware structure, the *thread table*, for tracking the relationship between forker and forkee threads to implement the synchronizing join operations. Our goal is a space-efficient structure that enables the implementation of fast thread management operations. We pursue this goal with a bookkeeping structure of limited size, that is for a bounded number of threads, so that thread creation, termination, and selection can be implemented with fast circuits within a bounded area of silicon real estate.

In the following, N shall be the number of hardware threads supported by our architecture. Furthermore, thread operations refer to hardware threads unless specified explicitly. For example, *thread creation* refers to allocating a hardware thread, and *thread termination* means releasing a hardware thread. We split the discussion of the proposed microarchitecture into three parts: (1) we introduce the hardware thread table, (2) we discuss the use of the link register to support an unbounded number of software threads with a bounded number of hardware threads, (3) we illustrate the function of thread table and link register by discussing two simple execution scenarios.

4.1 Thread Table

Figure 8 shows the organization of the thread table for $N = 4$ hardware threads T0–T3. Each hardware thread consists of a state field, a program counter, an identifier for a blocking thread, and base and limit addresses of the runtime stack. In addition, we maintain a 2-dimensional table of $N^2 - N$ join bits, one for each pair of forker and forkee thread. The *join bit* records whether *the forkee is active*, that is whether it has executed (join-bit value 0) or has not (join-bit value 1) executed the corresponding `join` instruction.

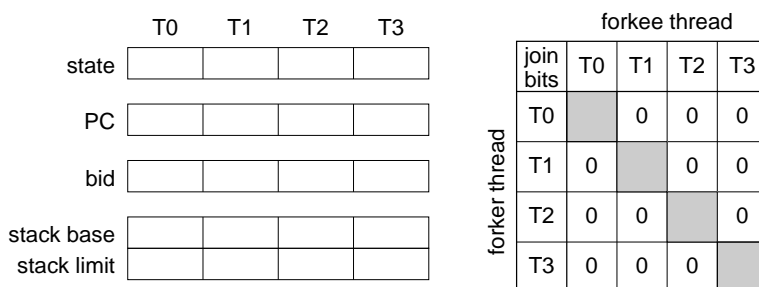


Figure 8: Thread table structure: for N hardware threads, the table contains $\Theta(N)$ bits for each threads’ state, program counter, an identifier of the blocking peer thread, and runtime stack addresses, plus $N^2 - N$ join bits for pairs of forker and forkee.

The set of states for a hardware thread include the following:⁵

unused The thread is not assigned to a software thread, and may not be scheduled for execution. Instead, it is available for shepherding a newly forked software thread.

active The thread is actively shepherding a software thread, and may be scheduled for execution.

⁵Additional states may be introduced in support of features such as atomic regions, for example.

join-blocked (applies to forker threads only) A forker thread has executed a join instruction, but the forkee has not executed the corresponding join instruction yet. The thread may not be scheduled for execution.

load-blocked The thread has issued a load instruction to memory, which has not responded yet. The thread may not be scheduled for execution.

load-commit The thread has an outstanding memory request, which has been serviced by the memory. The thread should be scheduled for execution to finalize the pending memory transaction.

States ‘load-blocked’ and ‘load-commit’ support a split load operation, and are described in more detail in Section 5.3.

The program counter (PC) of a hardware thread in Figure 8 contains the memory address of the next instruction to be executed. Our architecture permits issuing an instruction of one thread per clock cycle. There is no context switch overhead across hardware threads. Just the opposite, the default mode of operation issues instructions from different threads during each clock cycle, as already implemented in HEP [24].

The blocking thread identifier field (**bid**) in Figure 8 is needed to enforce the forker-continues invariant. This field stores the thread identifier of a thread’s forkee, in case the forker thread executes the **join** instruction before the forkee. For example, if thread T0 forks thread T1, and T0 executes its **join** instruction before forkee thread T1 reaches the corresponding **join** instruction, forker T0 must block until forkee T1 reaches the **join** instruction. Should forker thread T0 fork more than one forkee thread, we must ensure that T0 is reactivated only when thread T1 reaches the **join** instruction. To that end, we record the thread identifier of the forkee in the **bid** field of forker thread T0.

The stack base and limit fields of the thread table in Figure 8 record the range of memory assigned to the runtime stack of each hardware thread. An operating system may initialize the range fields when booting the processor. Each hardware thread obtains a private runtime stack as scratch memory for the software threads it shepherds. Typically, the runtime stack is used for local variables of functions called by a software thread, including those called due to a degraded fork attempt.

The join-bit table in Figure 8 records the activity of a forker’s forkee threads. This table can be implemented as an $(N \times N)$ -bit SRAM, for example. Each row is associated with a forker thread. If a forkee is active and has not executed the corresponding **join** instruction yet, the join bit is assigned value 1, otherwise value 0. The join-bit table enables us to reuse forkee threads if they join before the forker executes the corresponding join, see Figure 6(b).

4.2 Extended Link Register Semantics

Reusing hardware forkee threads can lead to the situation where a potentially unbounded number of **join** statements are yet to be executed by an active forker thread while the corresponding forkee threads have long terminated. Figures 5 and 6 above illustrate this case, where a single forker thread T0 creates an arbitrarily large number of n forkees before it executes the corresponding n join statements. We need to record the information about success or failure of the fork instructions to enable proper interpretation of the corresponding join instructions by the forker thread. Since managing an unbounded amount of state is generally more efficient in software than in hardware,

we wish to pass this information from hardware to software. To that end, we employ an established mechanism, the *link register*, and extend its use to support fork and join instructions. The software is responsible for spilling the link register on the runtime stack.

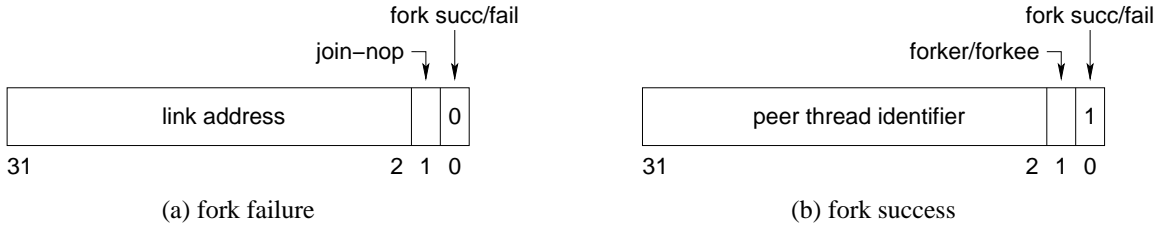


Figure 9: Use of link register in case of an unsuccessful (a) and successful (b) fork attempt. This example assumes 32-bit words, with instructions being aligned to 32-bit boundaries, such that the least significant two bits do not encode address bits.

The `fork` instruction generates the contents of the link register as a side effect, analogous to a jump-and-link instruction [15]. The information assigned by the `fork` instruction is needed for interpreting the associated `join` instructions, just like the returning jump uses the link address in case of a function call. We need to pass three pieces of information from a fork to the associated joins, as illustrated in Figure 9: (1) One bit enables us to distinguish between a successful or unsuccessful fork. (2) If the fork is unsuccessful and degrades into a function call, the remaining bits of the link register shall contain the conventional link address. The `join-nop` bit is initialized to zero. When interpreting the corresponding `join` instruction in the callee as a return statement we toggle the `join-nop` bit. Then, during execution of the corresponding `join` instruction in the caller, we test for value one to determine whether we must interpret the `join` as a `nop`. (3) If the fork succeeds, the architecture creates two link register values, one for the forker and one for the forkee thread. One bit identifies the thread as forker or forkee, and the remaining bits encode the peer thread identifier. The peer thread identifier associated with a forker is the forkee identifier and vice versa. Together, the identifier of a running thread and the identifier of the peer thread in the link register facilitate selection of the associated join bit in the join-bit table.

link register fields			description
succ/fail	fkr/jnop	peer tid/link addr	
succ	forker	forkee tid	peer is forkee tid
succ	forkee	forker tid	peer is forker tid
fail	0	link address	return to link address
fail	1	—	interpret join as nop

Table 1: Extended link register semantics.

Table 1 summarizes the four usage cases of the link register including assignments to the individual register fields that form the *link register triple*. The fork success/fail field and the forker/forkee field, also used as `join-nop` field, require one bit each. As illustrated in Figure 9, we might use the least significant bits of a 32-bit, big-endian architecture with 32-bit alignment of instruction words to store these two fields, because these two byte-selector bits are typically unused anyway.

The following pseudo-assembly code demonstrates the use of the link register in the presence

of two nested forks. When function `fork-foo-bar` is entered, the link register shall hold its return address.

```

fork-foo-bar:
  sub  sp sp 8    # create stack frame
  st   lr 0(sp)  # spill lr for return

  fork foo       # first fork (assigns lr)
  st   lr 4(sp)  # spill lr for join with foo
  fork bar       # second fork (assigns lr)

      :

  join lr        # join bar
  ld   lr 4(sp)  # restore lr for join with foo
  join lr        # join foo

  ld   lr 0(sp)  # restore link register
  add  sp sp 8   # destroy stack frame
  jr   lr        # return

```

In this code fragment the link register is used for three purposes: (1) to pass the return address of `fork-foo-bar` to the returning jump at the end of the program, (2) to pass the link information generated by the first fork instruction to the corresponding join, and (3) to pass the link information of the second fork instruction to the corresponding join. We need to spill the link register value twice onto the runtime stack, first to save the return address before the fork overwrites this value, and second to save the value generated by the first fork instruction before the second fork instruction overwrites that value. Note that the fork/join pairs for `foo` and `bar` are nested. Thus, we do not need to spill the link register between instruction `fork bar` and the subsequent `join lr`, assuming the program contains no further function calls or forks between these instructions. The use of the link register in support of fork/join pairs is compatible with the use for function call/return pairs, including common conventions for function calls and register spilling.

4.3 Execution Scenarios

We discuss three multithreaded program executions to illustrate the basic use of the thread table and link registers as introduced in Sections 4.1 and 4.2. We assume that the thread table comprises four threads, and that hardware thread `T0` shepherds execution of the initial software thread of a program.

Execution Scenario 1

Figure 10 illustrates the scenario where hardware thread `T0` forks a first software thread that is mapped to hardware thread `T1`. While thread `T1` is active, thread `T0` forks another software thread, which is mapped to hardware thread `T2`. The fork tree structure of this multithreaded program is shown in Figure 10(a). The thread diagram in Figure 10(b) includes the link register triples generated for the forker and forkee threads. Figure 10(c) shows the state transitions of the relevant portions of the thread table as it transitions due to fork and join events. For each step, the table depicts the state after committing the transitions due to the instructions shown below the diagrams,

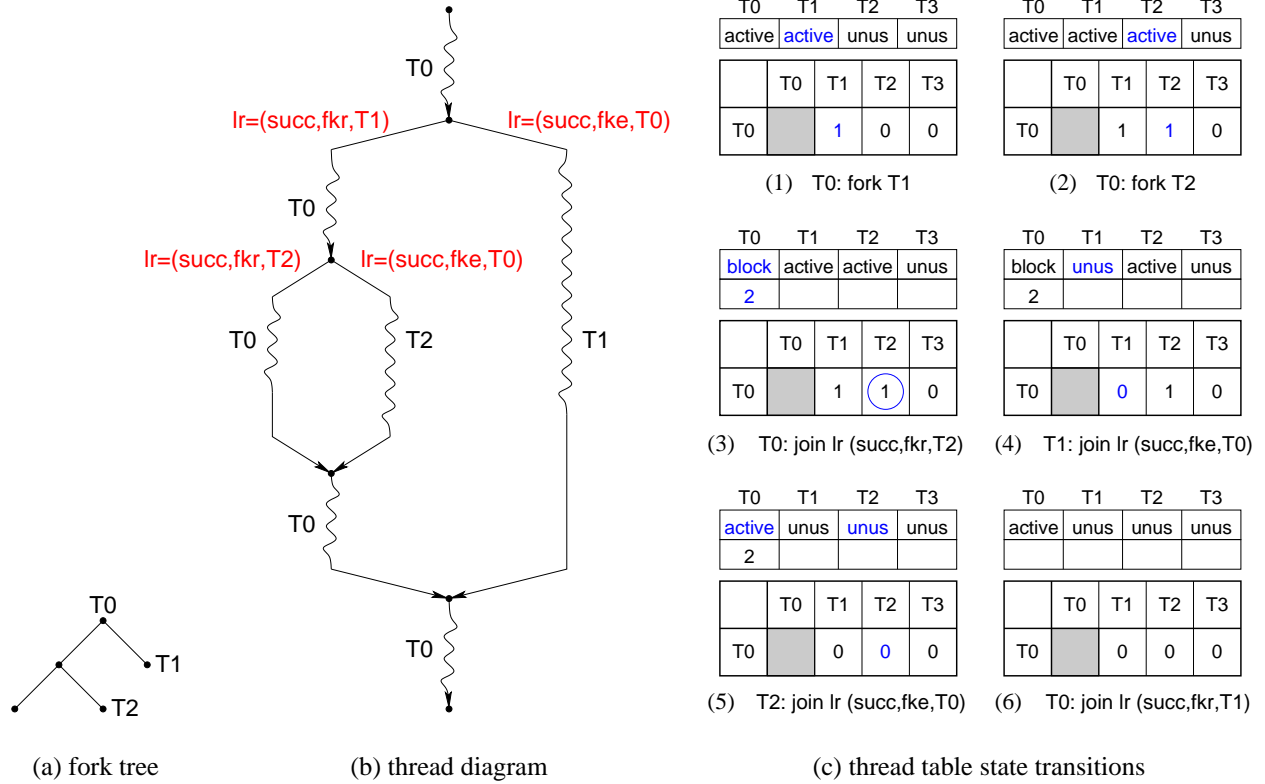


Figure 10: Basic execution scenario of a multithreaded program. We only show the thread table row of forker thread T0.

with the shepherding thread identifier and, in case of joins, the link register triple. Initially, we assume that thread T0 is the only active hardware thread, the state of all other threads shall be ‘unused,’ and all join bits are initialized to 0.

The first event during execution is the fork performed by hardware thread T0, see step (1) of Figure 10(c). We assume that our thread table management hardware detects that thread T1 is unused, and the fork is successful. The link register value passed to thread T0 asserts the fork success bit, the forker bit since T0 is a forker thread, and records thread T1 as forkee thread. The link register value generated for forkee thread T1 asserts the fork success bit, marks the thread as forkee, and assigns thread T0 as forker thread. During the second successful fork event, step (2), the link register values are assigned analogously. At this point in time, the thread table contains three active threads T0, T1, and T2. Forker thread T0 has two active forkee’s T1 and T2, which is reflected by the join bits in step (2) of Figure 10(c).

In step (3) thread T0 executes a join instruction. Thread T0 is the first of the two peers, forker T0 and forkee T2, to attempt synchronization. The link register identifies thread T0 as forker with forkee peer T2. This information facilitates retrieving the (encircled) join bit in row T0 and column T2. Since the join bit has value 1, the forkee is still active. According to the forker continues invariant, forker T0 must block until forkee T2 executes the corresponding join statement. We switch the state of thread T0 to ‘block,’ and record identifier 2 of blocking thread T2 in the bid field of T0.

In step (4) thread T1 executes a join instruction. The link register identifies thread T1 as forkee with forker peer T0. We terminate thread T1 by assigning state ‘unused,’ and toggle the associated join bit to value 0. Thread T0 remains blocked. In step (5) thread T2 joins. According to the link register, we identify T2 as a forkee, which allows us to terminate T2 by assigning state ‘unused,’ and toggling the join bit in the row of forker T0. Furthermore, thread T2 blocks thread T0, as recorded in the `bid` field of T0. Consequently, forker T0 may continue execution. We reactivate thread T0 by assigning state ‘active.’ Thread T0 executes the last join instruction in step (6). It joins with forkee thread T1. Since the associated join bit is 0, we deduce that T1 has terminated already. Thus, forker T0 continues execution without changes to the thread table.

Execution Scenario 2

Figure 11 illustrates a threaded execution scenario with reuse of forkee threads. In contrast to Scenario 1, the execution time of thread T1 shall be so short that it terminates before thread T0 executes its second fork statement.

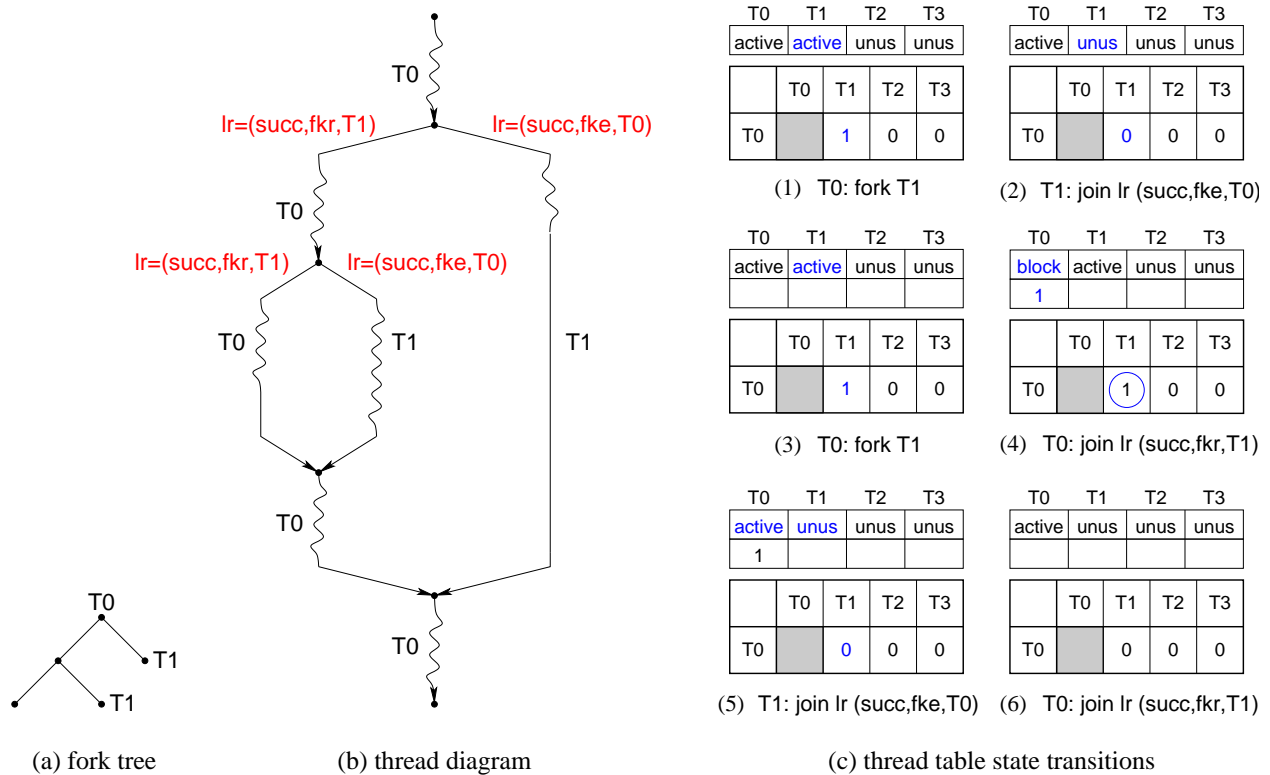


Figure 11: Execution scenario of a multithreaded program with thread reuse.

Step (1) of Scenario 2 is the same than in Scenario 1. In step (2), thread T1 performs a join. Since T1 is a forkee, we terminate T1 by reassigning ‘unused’ to its state, and toggling the join bit to value 0. Note that the state of the thread table is identical to the initial state before the first fork. In step (3) thread T0 forks a second software thread. Since thread T1 is unused, we reuse T1 to shepherd the new forkee of T0. We record the mapping by assigning state ‘active’ to thread T1 and toggle the join bit to value 1. The thread table is now in the same state than after the first

fork event. The difference in event history is encoded in the link register values.

In step (4) thread T0 joins. The link register identifies T0 as forker and the corresponding forkee as T1. Since the associated (encircled) join bit has value 1, T1 is still active, and we block thread T0. We record T1 in the `bid` field of T0. In step (5) thread T1 joins. According to the fork structure, this join corresponds to the second fork of thread T0. Since the link register value identifies T1 as forkee, we terminate T1. Furthermore, we reactivate forker thread T0 which has been blocked waiting for T1. Finally, in step(6) thread T0 joins with forkee thread T1, which did terminate already. Thus, thread T0 continues execution without modifications to the thread table.

Note that the reuse of thread T1 is not recorded in the thread table at all. Instead, the thread table records at each point in time which hardware threads are active forkees. The fact that hardware threads are reused is encoded in the link register values, which the software must spill on the runtime stack to support nested fork structures.

Execution Scenario 3

Figure 12 illustrates an execution scenario with a failed fork. Since a failed fork does not modify the thread table, the state transitions associated with the successful forks resemble those of the previous scenarios, and we omit the derivation of the state transitions of the thread table. The fork tree in Figure 12(a) illustrates the underlying fork structure. Thread T0 forks thread T1 and subsequently thread T2. Thread T1 forks thread T3. At this point the four hardware threads of the machine are in use. When thread T3 forks another software thread, the mapping into a hardware thread fails, and T3 executes the forked software thread by degrading the fork into a function call.

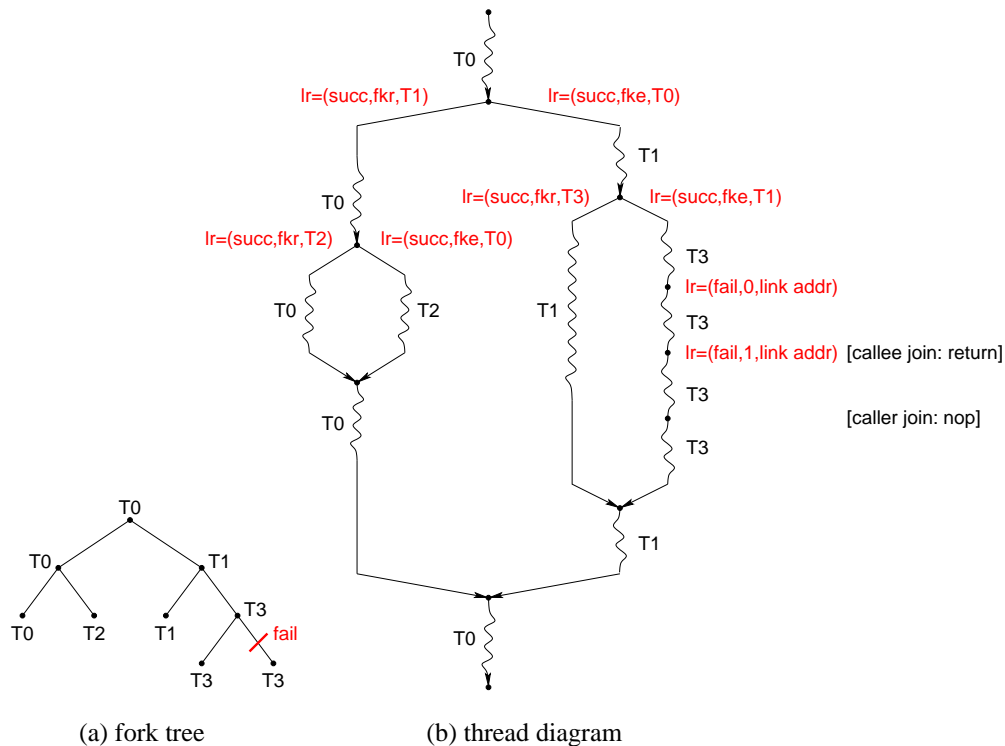


Figure 12: Execution scenario of a multithreaded program with a failed fork.

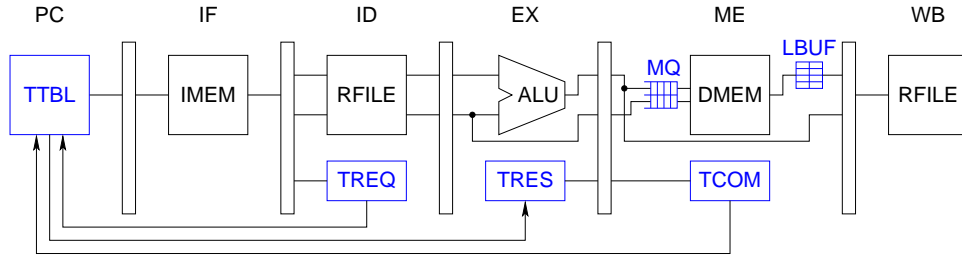


Figure 13: Sketch of a multithreaded RISC pipeline with thread table (TTBL) and thread modules (TREQ, TRES, TCOM), and with memory queue (MQ) and load buffer (LBUF) for decoupling the data memory (DMEM).

The thread diagram shows the link register values exported by the hardware to software. The interpretation of the link register values assigned by the successful forks proceeds analogous to those discussed in Scenarios 1 and 2. Therefore, we limit our discussion to the case of the failed fork.

When thread T3 executes its `fork` instruction, all four hardware threads of our machine are active already. The fork fails because no more hardware threads are available to shepherd the new software thread. Hence, the fork degrades into a function call, and the newly assigned link register of thread T3 encodes the failure as well as the return address for the callee. Thread T3 proceeds shepherding the callee until it reaches the callee’s `join` instruction. The processor identifies this `join` as a returning jump, because of the fail bit in the link register and because the `join-nop` bit is zero. As a side-effect of the `join` instruction, the processor replaces the `join-nop` bit in the link register with value one as a preparatory step for the interpretation of the next (in temporal order) `join` instruction. After the returning jump, thread T3 continues shepherding the execution of the caller which ends in a `join` instruction. The link register indicates a failed fork and contains a one in the `join-nop` field. Therefore, the hardware interprets the caller `join` as a `nop`.

Note that no bookkeeping is required in the thread table to cope with an unsuccessful fork attempt. The thread table is merely inspected by the `fork` instruction to identify that no hardware thread is available for shepherding a new software thread.

5 Multithreaded Processor Microarchitecture

In this section, we extend the pipelined RISC architecture of Hennessy and Patterson [15] into a multithreaded processor with implicit granularity adaptation. The choice of this simple architecture allows us to focus on our new ideas while avoiding unnecessary complexity. It should be possible to incorporate implicit granularity adaptation into various different architectures in principle, although detailed design work is needed to examine the cost effectiveness on a case-by-case basis.

Figure 13 sketches the extended 6-stage pipeline. The vanilla RISC datapath consists of five pipeline stages IF, ID, EX, ME, and WB. We add the PC-stage in front of the IF-stage to accommodate the thread table and a thread scheduler. Control modules TREQ, TRES, and TCOM support the implementation of fork degradation. We decouple the data memory interface from the pipeline to support one outstanding load request per thread. Furthermore, the register file shall have one segment per hardware thread. In the following, we outline the operational aspects of fork degradation, thread scheduling, and a possible implementation of memory latency hiding.

5.1 Fork Degradation

We implement fork degradation with three structures: (1) thread table and scheduler of the PC-stage, (2) thread modules `TREQ` in the ID-stage, `TRES` in the EX-stage, and `TCOM` in the ME-stage, and (3) two link registers `LRR` and `LRE`.

Thread Table

As discussed in Section 4.1, the thread table (TTBL) maintains the state of hardware threads. In particular, it records thread creation and termination by means of `fork` and `join` instructions. The thread table receives fork and join requests from modules `TREQ` and `TCOM`. Upon receipt of a *fork request*, it scans the state fields of the individual threads in search of an ‘unused’ thread. If an unused thread exists, the fork request is successful, and the thread table responds with an unused thread identifier. Otherwise, the thread table responds with a failure code.

When the thread table receives a *join request*, it must terminate or block the joining thread. As described in Section 4, we use the join table and the `state` and `bid` fields of the thread table to record the relationship between forker and forkee threads. After receiving a join request, the thread table inspects these fields including the join table entry as determined by the link register. It blocks a joining thread by assigning state ‘blocked’ and terminates a joining thread by assigning state ‘unused.’ Also, blocked peer threads are reactivated if the identifier of the joining thread matches the entry in the `bid` field.

Thread Modules

Thread modules `TREQ`, `TRES`, and `TCOM` spread the interactions of the pipeline with the thread table across multiple clock cycles. We split the `fork` instruction across three pipeline stages while the `join` instruction remains confined to the ME-stage.

The `TREQ` module is located in the ID-stage, where it identifies `fork` instructions by opcode. When a `fork` instruction enters the pipe, the `TREQ` module signals a fork request to the thread table. During the clock cycle following a fork request, the thread table responds with a fork success or fail signal. Also, in case of a successful fork, the signal is accompanied by a new forkee thread identifier. The `TRES` module in the EX-stage forwards the reply from the thread table to the ME-stage, if the instruction occupying the EX-stage is a fork instruction.

Yet one clock cycle later, when the `fork` instruction occupies the ME-stage, the `TCOM` module commits the `fork`. In case of a successful fork, it signals the thread table to activate the new forkee. Otherwise, no action is required. The `TCOM` module also composes the link register triples for a successful fork, as explained below. If a `join` instruction reaches the ME-stage, the `TCOM` module signals a join request to the thread table, which includes forker and forkee thread identifiers.

Link Registers

We assume that each hardware thread reserves one register in its segment as a link register. As described in Section 4, the link register passes the information from a `fork` instruction to the associated `join` instructions, coupling the interpretation of the joins to the success of the fork. The detour from the `fork` instruction through the link register, and potentially via register spilling through the runtime stack back to the `join` instructions, supports an unbounded number of software threads.

In case of a regular function call or an unsuccessful fork, only one link register (**LRR**) is needed to store the link address, because the control flow remains within the context of the shepherding hardware thread. We use the traditional link register mechanism for this purpose. The link address is computed in the ID-stage, and is passed through the EX and ME stages before it is written back into the register file.

In case of a successful fork, control flow splits into two threads. Now, we have to pass the fork information to both hardware threads the forker and the forkee. We denote the link register of the forker as **LRR** and the link register of the forkee as **LRE**. The **TCOM** module is responsible for generating the link triples for both forker and forkee threads. Link register **LRR** holds the link register triple of the forker thread. It contains the fork success bit and the forkee thread identifier both forwarded by the **TRES** module. The forkee triple in the **LRE** register contains the success bit and the thread identifier of the forker, and is passed to the forkee thread. During the write-back phase, both link register values are stored in the link registers of the segments associated with the forker and forkee threads, respectively.

Fork Walk-Through

In the following, we describe the traversal of a **fork** instruction through the processor pipeline. We assume that the thread scheduler selects an active hardware thread, whose program counter (**PC**) is issued to the instruction fetch (**IF**) stage, and the instruction memory returns the **fork** instruction. It is decoded in the ID-stage. Simultaneously, the **TREQ** module identifies fork instructions by opcode, and signals a fork request to the thread table.

When the **fork** instruction occupies the EX-stage, the thread table responds to the **TRES** module. If a hardware thread is available for shepherding the forked software thread, the thread table reserves the forkee thread and responds with its thread identifier. Otherwise, if all threads are active, the response indicates an unsuccessful fork. The **TRES** module relays the response of the thread table to the ME-stage.

The **TCOM** module commits the fork. If the fork request is successful, the **TCOM** module signals the thread table to commit the reserved forkee thread, and initializes the link register values for the forker and forkee in the **LR** and **LRE** portions of the ME pipeline register. In case of an unsuccessful fork request, the **TCOM** module effects the degradation of the **fork** instruction into a function call.

We place the **TCOM** module in the ME-stage of the pipeline, because this is the stage where the RISC pipeline commits an ordinary function call by feeding the address of the function entry point back to the program counter (**PC**). When the multithreaded processor executes a **fork** instruction, the ALU computes the same program counter as for an ordinary function call. However, the **TCOM** module directs the thread table to consume the program counter in one of two ways. In case of a successful fork, the program counter is stored in the **PC** field of the forkee thread. In contrast, if the fork fails, the program counter is stored in the **PC** field of the forker thread, which will subsequently jump to the function as would be the case with an ordinary function call.

5.2 Thread Scheduling

Our processor architecture enables context switching amongst hardware threads during each clock cycle. The thread scheduler is responsible for selecting an active thread in the thread table, and supplies its program counter to the IF-stage. Unused and blocked threads are not eligible for

execution. A simple scheduling algorithm such as a round-robin scheduler guarantees fairness, and can be based on a cyclic prefix circuit.

The datapath in Figure 13 shows a simple datapath without any interlocks and forwarding paths. This design relies on the thread scheduler to select each thread only as often as required to prevent data and control hazards. Alternatively, we could invest into a more complex datapath with interlocks and forwarding. The former choice facilitates a simple, faster hardware design at the expense of allowing a single thread to be scheduled during every third or fourth clock cycle only. In contrast, the latter choice invests hardware complexity to improve the performance of single-threaded programs [19]. We may implement fork degradation for either of these choices.

5.3 Memory Latency Hiding

A primary purpose of multithreading is the hiding of memory latencies. Although integrating memory latency hiding into the processor architecture is orthogonal to the implementation of fork degradation, it does impact the design of the thread scheduler. Therefore, we discuss this topic as far as it relates to our proposal.

Figure 13 emphasizes the structures surrounding the data memory (DMEM). To prevent load instructions from stalling the pipeline in face of high memory latencies, we decouple the data memory from the pipeline by introducing a *memory queue* (MQ) and a *load buffer* (LBUF). The memory queue is used to enqueue load and store instructions, and the load buffer stores load values returned by the data memory. The load buffer has one entry per hardware thread, so that each hardware thread can have one outstanding load request. This design is independent of the memory subsystem, which may include caches, employ memory banks, be a pipelined memory architecture, or be distributed across a larger machine. Although we describe the latency hiding implementation in the context of data memory, the same decoupling can be applied to the instruction memory (IMEM).

We illustrate the interaction between the thread scheduler and the decoupled data memory by means of the design of a *split load* instruction. The split load instruction shall not be part of the instruction set. Instead, we maintain the regular load instruction but implement the instruction such that the hardware interprets the load as a split load. As a concrete example, assume we have a regular load instruction for a RISC pipeline:

```
lw r9 4(r8)
```

which loads into register `r9` the word stored at the effective address computed by adding immediate value 4 to the value stored in register `r8`. We split this instruction into two phases, the *load issue* and the *load commit* phase to match the organization of the decoupled memory:

```
lw lbuf[tid] 4(r8)      # load issue
lw r9 lbuf[tid]        # load commit
```

The load issue phase enqueues a tuple consisting of thread identifier `tid` of the shepherding hardware thread and the effective address in the memory queue. After the memory has serviced the load request, it places the loaded value into the field of the load buffer associated with thread `tid`. Thereafter, the load commit phase reads the value from the load buffer and completes the load by writing the value back into register `r9`.

The execution of the two phases requires interaction between the thread scheduler and the data memory as follows: when a load instruction traverses the pipeline for the first time, it enters

the load issue phase. Upon enqueueing the load request into the memory queue, we assign state ‘load-blocked’ to the shepherding thread, cf. Section 4.1. The load instruction passes through the WB-stage without stalling the pipeline as if it were a `nop`. The shepherding thread will not be scheduled for execution until the data memory places the loaded value into the load buffer. The load buffer signals this event to the thread table. In response, we change the state of the thread to ‘load-commit,’ cf. Section 4.1.

The thread scheduler may now select the thread for execution, and reissue the original load instruction, this time in order to commit the load. During the load commit phase, the load instruction passes through the pipeline until it reaches the ME-stage. There, it reads the loaded value from the load buffer and passes it to the WB-stage, where the value is written back into the register file in the same fashion a regular load instruction would be implemented. At this point in time, the execution of the load instruction is complete. The thread state can be reset to the state before the load instruction has been issued for the first time, commonly state ‘active.’

6 Related Work

Implicit granularity adaptation has been studied in the context of the multithreaded programming languages Mul-T [18, 20] and Cilk [7, 12]. Both language implementations are based on *lazy task creation* [20], and both languages maintain task dequeues (doubly-ended queues) of activation frames in software. One way of characterizing the overhead of these implementations is to compare the runtime of a multithreaded program executed on one single-threaded processor with the fastest sequential version. The experimental evaluations in [20] and [12] show that these runtimes range from a factor close to 1 up to about 2 for different suites of benchmark programs. Cilk has a performance advantage over Mul-T, which is due to its compilation strategy. The Cilk compiler generates a fast and a slow clone for each Cilk procedure, as described in [12, Section 4]. The work on *indolent closure creation* [27] revealed that a different compilation strategy can reduce the runtime overhead of a failed fork (spawn) even further.

Even the smallest overheads due to the parallel runtime system matter in practice, because amortizing a loss of a seemingly harmless factor of 2 in the sequential runtime requires doubling the number of processors to achieve the same speedup that an ideal parallel runtime system could provide with half the number. This is the gap where our multithreaded processor helps: it reduces the overhead of both forking a thread and degrading a fork into a function call to zero. The hardware manages the bookkeeping of threads without the need for software contexts.

The performance benefit we can expect from implicit granularity adaptation depends on the underlying architecture. The simple pipeline described in Section 5 offers a potential performance benefit by means of memory latency hiding. More complex architectures may also exploit instruction level parallelism. Independent of the base architecture, however, implicit granularity adaptation simplifies the programming of parallel applications. This is the primary benefit of our proposal: When using Cilk (or similar languages like Mul-T) for work-stealing across multiple multithreaded processors of a parallel machine, the same Cilk program exploits processor-level parallelism across the machine, and within each multithreaded processor it exploits memory latency hiding by interleaving multiple threads to speedup the execution. One multithreaded Cilk program can exploit both architectural structures seamlessly. Future research will have to determine whether hardware support is desired to reduce the cost of stealing a thread from a multithreaded processor. Irrespectively, since the Cilk programmer does not have to be concerned with how many

multithreaded processors with how many hardware threads a particular machine provides, such Cilk programs can be considered portable.

Memory latency hiding by means of a multithreaded processor is important with respect to the design of the memory system. Unlike depth-first scheduled threads, which can share a common cache quite effectively [6], Cilk threads tend to operate on distinct working sets. Hence, the memory size of a multithreaded processor that supports Cilk should be proportional to the number of hardware threads. Since a larger memory has longer latencies, the multithreaded processor should be able to hide these memory latencies. In contrast, hiding memory latencies by means of multithreading is known to be not scalable in parallel machines [5].

A large body of software implementations of multithreaded languages has been explored in the past. The papers on Lazy Threads [13] and Concert [17] contain a plethora of references to related work. Lazy threads are based on compiler support to customize the memory management of activation frames with so-called stacklets. This customization enables a more general handling of activation frames in the context of non-strict languages than is required by the semantics of a sequential function call. The Concert system 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. It is the compiler's responsibility to determine whether a parallel call can be replaced by a sequential function call.

Besides the application-centric scheduling goal for a single multithreaded application, preemptive threads have been introduced with a processor-centric scheduling goal, which is to utilize the expensive processor resource. The most widely known types of preemptive software threads are user-level and kernel-level threads [23, 26, 21, 11]. The primary focus of preemptive threads is to enable the user or programmer of a machine to specify a number of independent threads that share a single or multiple processors. If one of these threads blocks, for example in a disk access, it is the responsibility of the scheduler to ensure utilization of the processor by switching to another thread. Combining the methods of scheduling multiple virtualized multithreaded applications on one or more multithreaded processors complicates the scheduling problem [25].

Multithreaded architectures appeared in early computer designs such as Bull's Gamma 60 [4], which used a primitive form of multithreading to hide the latency of all machine operations, including arithmetic, memory accesses, and I/O. Later designs emphasized the use of multithreading for *memory latency hiding* in multiprocessors, where memory access latencies are fundamentally large because they are dominated by communication distances. Burton Smith pioneered the use of multithreading for memory latency hiding in multiprocessors. He architected HEP in the late 1970's [24], later Horizon [29], and more recently Tera [2]. Due to today's microtechnologies, even single-processor architectures suffer from the so-called *memory wall* [33]. Multithreading has been applied in various facets; [31] provides a comprehensive review of multithreaded architectures. *Simultaneous multithreading* [30] is the most popular variant for today's superscalar processors. We are not aware of any existing architectural support for implicit granularity adaptation by means of fork degradation. While we do not foresee any principle difficulties, future work will have to show whether fork degradation can be implemented cost effectively in various computer architectures.

7 Conclusion

We propose a multithreaded processor architecture with implicit granularity adaptation as a means of interpreting programs with excess parallelism efficiently. We introduce fork degradation to map an unbounded number of software threads into a bounded number of hardware threads. Our architecture supports Cilk or Mul-T programs that specify the parallelism inherent in an application by executing these programs space and time efficiently.

Acknowledgements

I thank Matteo Frigo and Ahmed Gheith for our discussions on the reuse of hardware threads, and their help with the patent application [28]. Furthermore, Jim Peterson's and Ram Rajamony's comments on an earlier proposal motivated the refined architecture presented here.

References

- [1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition, 1996.
- [2] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton J. Smith. The Tera Computer System. In *4th International Conference on Supercomputing*, pages 1–6. ACM Press, 1990.
- [3] Arvind, David E. Culler, and Gino K. Maa. Assessing the Benefits of Fine-Grain Parallelism in Dataflow Programs. In *Supercomputing'88*, pages 60–69, Orlando, FL, November 1988.
- [4] M. Bataille. Something Old: The Gamma 60, The Computer that was Ahead of Its Time. *Honeywell Computer Journal*, 5(3):99–105, 1971.
- [5] Gianfranco Bilardi and Franco P. Preparata. Horizons of Parallel Computation. *Journal of Parallel and Distributed Computing*, 27(2):172–182, June 1995.
- [6] Guy E. Blelloch and Phillip B. Gibbons. Effectively Sharing a Cache Among Threads. In *16th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 235–244, Barcelona, Spain, June 2004.
- [7] 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.
- [8] Melvin E. Conway. A Multiprocessor System Design. In *Fall Joint Computer Conference*, pages 139–146. AFIPS, Spartan Books (vol. 24), October 1963.
- [9] David E. Culler, Seth C. Goldstein, Klaus E. Schauer, and Thorsten von Eicken. TAM—A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, 18(3):347–370, July 1993.
- [10] Jack B. Dennis and Earl C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Communication of the ACM*, 9(3):143–155, March 1966.

- [11] Ralf S. Engelschall. Portable Multithreading: The Signal Stack Trick for User-Space Thread Creation. In *Usenix Annual Technical Conference*, pages 239–250, San Diego, CA, June 2000.
- [12] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998.
- [13] Seth C. Goldstein, Klaus E. Schauser, and David E. Culler. Lazy Threads: Implementing a Fast Parallel Call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, August 1996.
- [14] Robert H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.
- [15] John Hennessy and David Patterson. *Computer Organization and Design*. Morgan Kaufmann, 2nd edition, 1998.
- [16] C. Antony R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, United Kingdom, 1985.
- [17] 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.
- [18] David A. Kranz, Jr. Robert H. Halstead, and Eric Mohr. Mul-T: a High-Performance Parallel Lisp. In *ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 81–90, Portland, OR, 1989.
- [19] James Laudon, Anoop Gupta, and Mark Horowitz. Architectural and Implementation Trade-offs in the Design of Multiple-Context Processors. In Robert A. Iannucci, editor, *Multithreaded Computer Architecture: A Summary of the State of the Art*, pages 167–200. Kluwer Academic Publishers, Boston, MA, 1994.
- [20] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy Task Creation: a Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, 1991.
- [21] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *PThreads Programming: A POSIX Standard for Better Multiprocessing*. O’Reilly, 1996.
- [22] Alexandru Nicolau and Joseph A. Fisher. Measuring the Parallelism Available for Very Long Instruction Word Architectures. *IEEE Transactions on Computers*, C-33(11):968–976, November 1984.
- [23] M. L. Powell, Steve R. Kleiman, Steve Barton, Devang Shah, Dan Stein, and Mary Weeks. SunOS Multi-thread Architecture. In *Winter 1991 USENIX Technical Conference and Exhibition*, pages 65–80, Dallas, TX, USA, 1991.
- [24] Burton J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. In *4th Symposium on Real Time Signal Processing*, pages 241–248. SPIE, 1981.

- [25] Bin Song. Scheduling Adaptively Parallel Jobs. Master's thesis, MIT Department of Electrical Engineering and Computer Science, January 1998.
- [26] Dan Stein and Devang Shah. Implementing Lightweight Threads. In *Summer 1992 USENIX Technical Conference and Exhibition*, pages 1–10, San Antonio, TX, 1992.
- [27] Volker Strumpfen. Indolent Closure Creation. MIT Laboratory for Computer Science, Technical Memo MIT-LCS-TM-580, June 1998.
- [28] Volker Strumpfen, Matteo Frigo, and Ahmed Gheith. Multithreaded Processor Architecture with Implicit Granularity Adaptation. United States Patent Application No. 11/101608, May 2005.
- [29] M. R. Thistle and Burton J. Smith. A Processor Architecture for Horizon. In *ACM/IEEE Conference on Supercomputing*, pages 35–41. IEEE Computer Society Press, 1988.
- [30] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-chip Parallelism. In *22nd Annual International Symposium on Computer Architecture*, pages 392–403, Santa Margherita Ligure, Italy, June 1995.
- [31] Theo Ungerer, Borut Robič, and Jurij Šilc. A Survey of Processors With Explicit Multithreading. *ACM Computing Surveys*, 35(1):29–63, 2003.
- [32] David W. Wall. Limits of Instruction-Level Parallelism. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, 1991.
- [33] William A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23(1):20–24, 1995.