LPTHREAD: A WORK-EFFICIENT THREAD MODEL FOR LOOP PARALLELIZATION

BY

THOMAS WEI-PING SOONG

B.S., University of Illinois at Urbana-Champaign, 2002

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2004

Urbana, Illinois

ABSTRACT

This thesis describes LPthread (Loop Parallelization Thread), a thread model designed to aid the process of exploiting thread-level parallelism in sequential programs through loop parallelization. The LPthread model provides a deadlock-free environment for exploiting loop parallelization by imposing thread creation constraints and by assigning thread priorities according to the serial program's control flow sequence. The spawn constraints allow the parallel program to fall back on the original sequential order and avoid deadlock. Moreover, the LPthread model provides an efficient infrastructure to resolve inter-thread dependencies. Communication between LPthreads has the same cost as a typical store instruction. Finally, the LPthread model is work efficient on parallelized sequential programs. The performance of an LPthread program with minimal inter-thread dependencies increases almost linearly with the number of processing units. To maximize performance, an LPthread can also be speculatively spawned.

To Mom and Dad

ACKNOWLEDGMENTS

Like my thesis, I will keep this section short and sweet. While I have dedicated this work to my parents, I will not attempt to pay tribute to them, since I can never fully express my gratitude for their infinite contributions and sacrifices. I can only use this opportunity to say, "Thank you."

While numerous people brought inspiration to my research, this thesis owes greatly to my adviser, Professor Matthew Frank, for his wisdom and patience. Being Matt's first research student to write a thesis, I hope to make him proud. As countless computer architects make their contributions toward faster computing, here is mine.

TABLE OF CONTENTS

1 INTRODUCTION

In the effort to achieve higher microprocessor performance, the exploitation of instruction-level parallelism (ILP) of sequential programs in dynamically scheduled superscalar processor has been the most dominant trend in both hardware [1]-[3] and software [4]-[6]. Nevertheless, the diminishing returns of wider instruction issue, as well as increasing on-chip memory have driven hardware and compiler designers to investigate more aggressive techniques for discovering program parallelism. Eventually all high-performance processors will exploit the benefits of thread-level parallelism, such as simultaneous multithreading (SMT) [7] or chip-scale multiprocessing (CMP) [8]. Since traditional ILP techniques are targeted for single thread execution, its benefit is very limited on multithreaded machines. While it is obvious that parallel application is needed for such architecture, automatic parallelization of general purpose programs is required for it to be successful. This thesis contributes a novel thread model that aids the process of exploiting thread-level parallelism in sequential programs through loop parallelization.

1.1 Finding Thread-Level Parallelism

Thread-level parallelism can be exploited by utilizing the fundamental property of control-flow independence in do-across loops. As demonstrated by Lam and Wilson's [9] and Rotenberg et al.'s [10] limit studies on control independence, substantial performance improvement can be gained. The mechanism of exposing thread-level parallelism in *N*-dimensional nested do-across loops is best explained by example.

```
do
  j = 0
  i = i + 1
  use1()
  y = use2(y)
  do
    j = j + 1
    use3()
    y = use4(y)
  while j < M
while i < N
```

Figure 1: An example of a doubly nested loop with nested parallelism, where concurrency can be exploited by simultaneously executing different `use1()` and `use3()` across different iterations.

Consider the doubly nested loop in Figure 1 that contains multiple loop-carried dependencies, and assume that each `use` subroutine requires substantial execution time. Sequentially, the next iteration of the outer loop cannot commence execution until its inner loop is completed, and the loop-carried dependent variables `i` and `y` are resolved. On the other hand, the data-independent subroutine `use1()` for the next iteration of the outer loop can be executed either as soon as the loop-back condition is satisfied, i.e., `i < N`, or speculatively. Therefore, the outer loop can be parallelized by having as many `use1()` subroutines in flight as possible. The same parallelization technique can be applied to the `use3()` subroutine within the inner loop. Therefore, the doubly nested loop is parallelized by simultaneously executing different `use1()` and `use3()` across different iterations while the iterations' loop-carried dependencies are being resolved. Furthermore, parallelism is extracted from all loop levels, and execution can be speculative. In essence, the multithreaded machine executes ahead of the sequential code by extracting and executing data-independent as well as control-independent instructions within the nested loop.

1.2 A Good Thread Model

Aside from extracting parallelism from all loop levels, a successful thread model for loop parallelization must overcome three challenges:

(1) The thread model must guarantee the parallelized program's forward progress.

(2) The thread model must provide efficient channel for interthread communication.

(3) The thread model must maintain load balance on multithreaded machines.

While the purpose of automatic parallelizing sequential programs is to maximize performance, program correctness remains the primary objective. Not only must the parallel program produce the same result as its sequential counterpart, it must also never deadlock, which is the most common predicament in parallel programming.

Since the compiler transforms all parallelized loop iterations into threads, loop-carried dependencies becomes interthread dependencies. The thread model must establish a low overhead communication channel to quickly resolve these dependencies that hinder parallelism.

To ensure that no processing unit's resource is going to waste while another processing unit is overstressed, the thread model's scheduling algorithm must also be work-efficient and maintain load balance on all its processing units. A parallel program is constrained by the *work* and *critical path* [11]. The work T_1 is the execution time of the original sequential program on one processing unit. The critical path T_8 is the ideal execution of the parallel program on infinite processing units and proportional to the height of the program's control dependence graph. The execution time of the parallel program executing on P processors with a work-efficient thread model is $T_p = O(T_1/P+T_8)$.

The Loop Parallelization Thread (LPthread) model was designed explicitly to exploit loop parallelism while meeting the three mentioned requirements. The LPthread model consists of four atomic operations that can be supported in hardware or emulated in software. The forward progress of an LPthread program is guaranteed as long as a minimum memory requirement, proportional to the sequential program's static loop depth, is satisfied in hardware. Meanwhile, software compiler analysis is needed to fully utilize the LPthread model in thread-level parallelism.

1.3 Related Work

Several different deadlock-free thread models have been developed for parallel computer architecture, such as the Provably Efficient Thread model [12] or the Cilk multithreaded language [11]. While these models are efficient for constructing general parallel applications, they were not designed to automate sequential code parallelization. Therefore, all programs using these models must endure the excruciating progress of parallelizing by hand. Meanwhile, parallel programs using the older Threaded Abstract Machine (TAM) [13] model must also be rigorously debugged to assure that they are deadlock-free.

On the other hand, several speculatively multithreaded architectures (such as Multiscalar [14] and The Superthreaded Architecture [15]) have thread models that also exploit control independence by pursuing multiple flows of control. These thread models can only exploit loop parallelism in one dimension; i.e., only one loop level's parallelism can be exploited. Therefore, their potential thread-level parallelism is limited. For example, Multiscalar is incapable of extracting much thread-level parallelism from the code in Figure 1. Assuming that the outer loop is parallelized, every iteration of the outer loop must wait for its previous iteration's inner loop to complete in order to resolve the loop-carried dependent variable `y`. Therefore, the parallel code becomes highly sequential. The same limitation is exhibited when the inner loop is parallelized. In contrast, the LPthread model overcomes these limitations by exposing thread-level parallelism in *N*-dimensional nested do-across loops.

2 WHAT IS AN LPTHREAD?

Like most threads, an LPthread is an independent stream of instructions that shares the process resources with other LPthreads. Aside from maintaining its own stack pointer, program counter, and registers, an LPthread also carries its own status bits and communication queues, as shown in Figure 2.

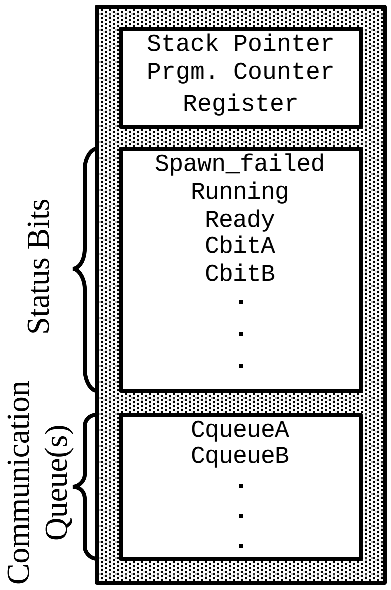


Figure 2: The structure of an LPthread. The communication queues are needed for interthread communication while the status bits are used to identify the thread's state.

2.1 Communication Queues

Any pending data from another thread that is necessary for execution is communicated and pushed into the communication queue (*Cqueue*) of the receiving thread. The number of Cqueues is arbitrary and allocated when the thread is first spawned. Each Cqueue is capable of communicating with only a single thread. Therefore, if a thread needs to receive data from two other distinct threads, a minimum of two Cqueues are required. A thread can only access the elements in a Cqueue in first-in first-out (FIFO) order. Finally, a thread can only access the data in its own Cqueue once the corresponding communication bit (*Cbit*) is set.

2.2 Status Bits

The three status bits - *Ready*, *Spawn_failed*, and *Running* - can only be manipulated by the scheduler and are used to determine the current state of the thread. The state transition diagram is shown in Figure 3.

- *Ready* bit: A thread's Ready bit is set to 1 only if it is ready for execution; otherwise, it remains 0. In other words, a thread's Ready bit is set to 1 only when all the interthread dependencies within its proceeding instructions are resolved.
- *Spawn_failed* bit: The Spawn_failed bit is initialized to 0 when the thread is first spawned and is only set to 1 if the current executing thread is unsuccessful in spawning a child thread.
- *Running* bit: When the Running bit is set to 1, it indicates that the thread is currently being executed. Conversely, a Running bit of 0 specifies that the current thread is idle.
- *Cbit*: With each allocated Cqueue there is a corresponding Cbit. The Cbit reflects the status of its Cqueue. By default, the Cbit is initialized to 0, and only the thread communicating with the corresponding Cqueue is able to set it to 1. As a rule, the Cbit should only be set to 1, if and only if its Cqueue has received all its messages. In other words, once the Cbit is set to 1, no more data can be stored into the Cqueue. All allocated Cbits form a FIFO structure. While the communicating thread(s) can write to each individual Cbit of the receiving thread. The receiving thread can only read its Cbits in FIFO fashion.

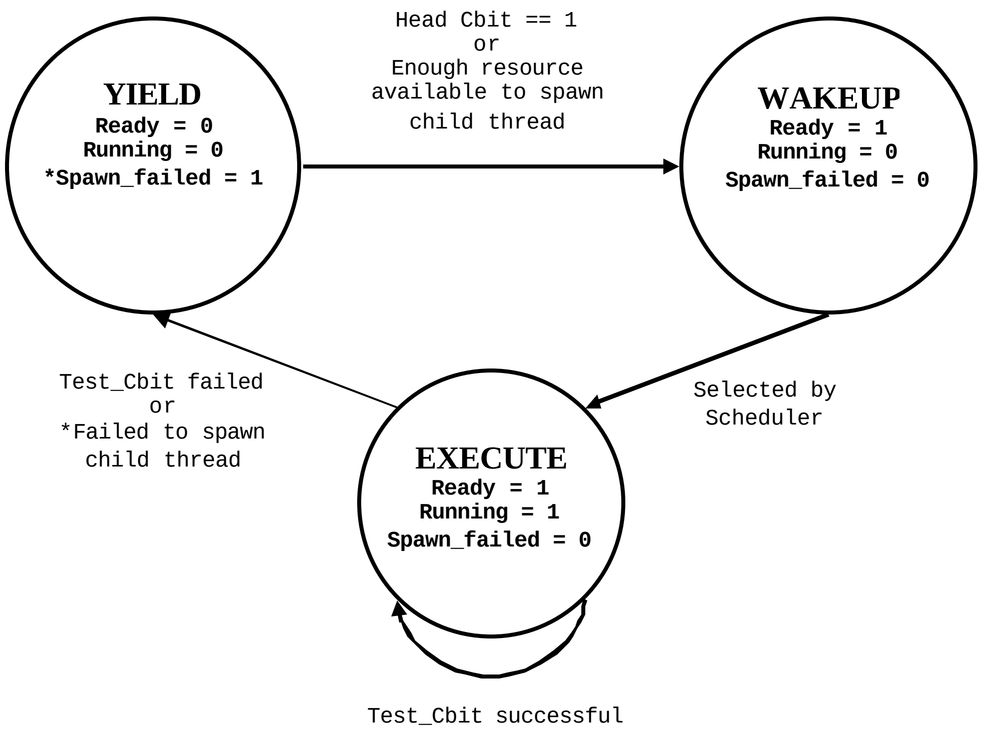


Figure 3: The state transitions of an LPthread. The outputs in each state correspond to how the scheduler will manipulate the thread's status bit during state transitions. The outputs on the edges represent the condition that would invoke a state change from the scheduler.

3 THREAD STATES

As shown in Figure 3, an LPthread has only three possible states: YIELD, WAKEUP, and EXECUTE. Only the scheduler can change the state of a thread.

1. YIELD: A thread can yield for three reasons. First, if the executing thread attempts to spawn a child thread and fails, its Spawn_failed bit is set to 1 by the scheduler and the thread yields. Second, an executing thread yields when its `test_Cbit` operation fails, since it currently does not have the required data to continue execution. Finally, the LPthread is initialized to be in the yielded state when it was first spawned. Hence, the yielded thread must have its head Cbit updated by an executing thread for it to wake up.
2. WAKEUP: Just as there are two ways to yield an executing thread, there are two way to wake up a yielded thread. All threads that are woken up have their Ready bit set to 1. First, a yielded thread with its Spawn_failed bit set to 1 will wakeup once the scheduler has determined that enough resources have been freed to wakeup the thread and retries its spawn of a child thread. Meanwhile the scheduler will reset the thread's Spawn-failed bit to 0 and set the Ready bit to 1. Second, a thread can wakeup when its Cbit is updated to 1. If an LPthread's Ready bit is already set to 1, then the LPthread has already woken up and the scheduler does nothing. Conversely, the schedule will wake up the LPthread by setting its Ready bit to 1, if the updated Cbit is at the head of the Cbit queue and the Spawn_failed bit is 0. Finally, a thread can also start from WAKEUP state when it was first spawned.
3. EXECUTE: As discussed later, the scheduler is responsible for selecting threads for execution fom the pool of thread in the WAKEUP state. Any thread that enters the EXECUTE state will have its Running bit set to 1. Meanwhile, a thread can either yield during its execution or retire once it is completed.

4 INTERFACE

The LPthread interface is a set of primitives that is used for both interthread communication and communication with the scheduler. Moreover, each primitive must be executed atomically without interruption. For explanation purposes, the interface is shown in C syntax.

4.1 LPthread* spawn(void*(*start_routine)(), int num_Cbits, bool start_state)

The spawn primitive is use to create a child thread and requires three parameters. If successfully spawned, the child thread is created to execute `start_routine`. Also, the number of Cqueues and Cbits allocated to the child thread will be equal to `num_Cbits`. The `start_state` parameter is used to initialize the initial state of the spawning child thread. When first spawned, a thread can be in the YIELD state, `start_state = 0`, or the WAKEUP state, `start_state = 1`. However, if `num_Cbits = 0`, the thread defaults to the WAKEUP state, since the spawning thread does not need to be synchronized.

As mentioned earlier, if a thread starts from the YIELD state, it implies that the LPthread requires data from another LPthread to commence execution. Therefore, the thread will only wake up once its pending data are received. On the other hand, a thread that is spawned to the WAKEUP state can immediately begin execution without waiting for any other thread. Hence, all the instructions from the beginning of the LPthread up until the first `test_Cbit` operation are all data-independent from any other spawned thread. Nevertheless, an LPthread initialized to the WAKEUP state does not suffer from deadlock even if the beginning instructions contain interthread dependencies. As mentioned later, the dependent LPthread would simply yield during its `test_Cbit` operation, where the depending LPthread will later wake it up.

Consequently, an LPthread is only initialized to the WAKEUP state as a performance optimization, not to maintain program correctness.

As a rule, a thread can only spawn one child thread for any level of a nested loop. Therefore, a thread executing a doubly nested loop can only spawn a maximum of two child thread. With one thread corresponding to the next outer loop while the other thread is the next inner loop. More importantly, the child threads' execution must correspond to the loop level at which they were created. In other words, if a thread is spawned at the outer loop level of its parent, it must execute the next iteration of the outer loop. As discussed in Chapters 6 and 7, this constraint is critical in ensuring that the LPthread model is deadlock-free.

If the spawn operation is successful, the scheduler will return a thread pointer to the parent thread, which contains the address of the child thread. The thread pointer is used for communicating with child thread. In effect, any subsequent thread that is spawn from the parent can have access to the thread pointer by having it passed in to its Cqueue. Hence, they will also be able to communicate with the child thread. Conversely, if the spawn fails due to insufficient memory, then the parent thread yields and will retry again once enough system resource has been freed up.

4.2 void update_Cbit(int Cbits_index)

The `update_Cbit` primitive sets the receiving thread's `Cbit[Cbits_index]` to 1. A receiving thread's Cbit should only be updated by the sender after all the messages have been stored into the correspond Cqueue of the receiving thread. Since all the messages in a Cqueue originate from a single sender, only the sender is authorized to manipulate the Cqueue's corresponding Cbit on the recipient.

4.3 void test_Cbit()

The `test_Cbit` operation reads the front element of its Cbit queue and is considered successful only when the front element is set to 1. If the head Cbit is 1, the scheduler dequeues the front element, and the calling thread continues execution. If the head Cbit is 0, the Cbit queue remains untouched. Furthermore, the calling thread must yield itself until its head Cbit is updated to 1. When the thread reenters the EXECUTE state, it will retry the `test_Cbit` operation. Hence, a thread can continue its execution only until its `test_Cbit` operation is successful. A thread can access its Cqueue only when the `test_Cbit` operation for the corresponding Cbit is successful, thus assuring that all messages arrived to the Cqueue. Since the `test_Cbit` only checks for the head Cbit, the Cqueues are accessed in the same sequence of its Cbits in the Cbit queue.

4.4 void stop()

The `stop` operation is executed only when a thread has completed its execution; therefore, it is the last operation that the calling thread will perform. Aside from deallocating the resources used by the current thread, this operation also signals the scheduler to assign the processing unit a new thread for execution. Moreover, if any of the yielded threads have its Spawn_failed bit set to 1, the scheduler will examine if the newly available resources are enough to wake up the oldest thread that experienced an unsuccessful thread spawn.

5 LPTHREADS IN ACTION

The mechanism for thread level parallelism using LPthreads is best explained by example. Consider the code in Figure 4; the code depicts an ideal case for loop parallelization, where almost no interloop data dependence exists. It is a triply nested loop where the number of iteration for each level of the loop is arbitrary. The only interloop dependencies are the iteration counters `i`, `j`, and `k`, which are used to identify the loops' end conditions. The subroutines `use1`, `use2`, and `use3` are completely data-independent and can execute in parallel when called. Therefore, significant parallelism can be found by concurrently executing as many `use1s`, `use2s`, or `use3s` as possible.

```
main()
{
  i = 0;
  do
    j = 0;
    use1()
    do
      k = 0;
      use2()
      do
        use3()
        k = k + 1
      while (k < c)
      j = j + 1
    while (j < b)
    i = i + 1
  while (i < a)
}
```

Figure 4: An example program with a triply nested loop. Significant parallelism can be found with LPthreads by concurrently executing as many `use1s`, `use2s`, or `use3s` as possible.

5.1 Exploiting Parallelism with LPthreads

Figure 5 shows how LPthread can transform the code in Figure 4 to extract thread level parallelism. Since the code consists of three do-while loops, any iteration of the outer loop will encounter at least one iteration of the middle loop and the inner loop. Similarly, an iteration of the middle loop will guarantee an iteration of the inner loop.

Let us first examine the `main` thread. As stated before, LPthread can only spawn one thread in each loop level. Hence, a maximum of three threads be can spawned in the `main` thread. To maximize parallelism, the `main` thread checks the end condition for each level of the loop while spawning up to three different threads `outer`, `middle`, and `inner`.

Assuming that variable `a` is greater than 1, the outer loop will execute more than once and the `main` thread will first spawn a thread of `outer`. Since the only inter-loop data dependency in the outer loop is the variable `i`, only one Cbit and Cqueue is needed by the child thread. Therefore, the `main` thread creates an LPthread that executes the next outer loop by doing `spawn(outer,1,0)`, which returns a pointer of the child thread and is stored in `thread_ptr1`. Since the `outer` child thread is initialized to the YIELD state, it cannot execute until `i` is received from the previous iteration.

The `main` thread must communicate with its `outer` child. The `main` thread stores the variable `i` in its child Cqueue through the thread pointer and updates its child's Cbit to wake up the thread. Similarity, the `main` thread spawns its `middle` and `inner` child threads if the values of `b` and `c` are greater than 1. After spawning all its children, the `main` thread proceeds to execute the `use` subroutines in the first iteration of each level of the triply nested loop. Hence, `use1`, `use2`,

```
main()
{
   i = 1
   j = 1
   k = 1
   if (i < a)
     thread_ptr1 = spawn(outer,1,0)
     thread_ptr1->Cqueue[0].enqueue(i)
     thread_ptr1->update_Cbit(0)
   if (j < b)
     thread_ptr2 = spawn(middle,1,0)
     thread_ptr2->Cqueue[0].enqueue(j)
     thread_ptr2->update_Cbit(0)
   if (k < c)
     thread_ptr3 = spawn(inner,1,0)
     thread_ptr3->Cqueue[0].enqueue(k)
     thread_ptr3->update_Cbit(0)
   use1()
   use2()
   use3()
   stop
}
```

```
outer()
{
  j = 1
  k = 1
  test_Cbit
  i = Cqueue[0].dequeue
  i = i + 1
  if (i < a)
    thread_ptr1 = spawn(outer,1,0)
    thread_ptr1->Cqueue[0].enqueue(i)
    thread_ptr1->update_Cbit(0)
  if (j < b)
    thread_ptr2 = spawn(middle,1,0)
    thread_ptr2->Cqueue[0].enqueue(j)
    thread_ptr2->update_Cbit(0)
  if (k < c)
    thread_ptr3 = spawn(inner,1,0)
    thread_ptr3->Cqueue[0].enqueue(k)
    thread_ptr3->update_Cbit(0)
  use1()
  use2()
  use3()
  stop
}
```

```
middle()
{
  k = 1;
  test_Cbit
  j = Cqueue[0].dequeue
  j = j + 1
  if (j < b)
    thread_ptr1 = spawn(middle,1,0)
    thread_ptr1->Cqueue[0].enqueue(j)
    thread_ptr1->update_Cbit(0)
  if (k < c)
    thread_ptr2 = spawn(inner,1,0)
    thread_ptr2->Cqueue[0].enqueue(k)
    thread_ptr2->update_Cbit(0)
  use2()
  use3()
  stop
}
```

```
inner()
{
  test_Cbit
  k = Cqueue[0].dequeue
  k = k + 1
  if (k < c)
    thread_ptr1 = spawn(inner,1,0)
    thread_ptr1->Cqueue[0].enqueue(k)
    thread_ptr1->update_Cbit(0)
  use3()
  stop
}
```

Figure 5: Using LPthread to exploit loop level parallelism from the code in Figure 4. Each block of code is an LPthread.

and `use3` are all executed once in the `main` thread. Finally, the `main` thread performs the `stop` operation before it retires.

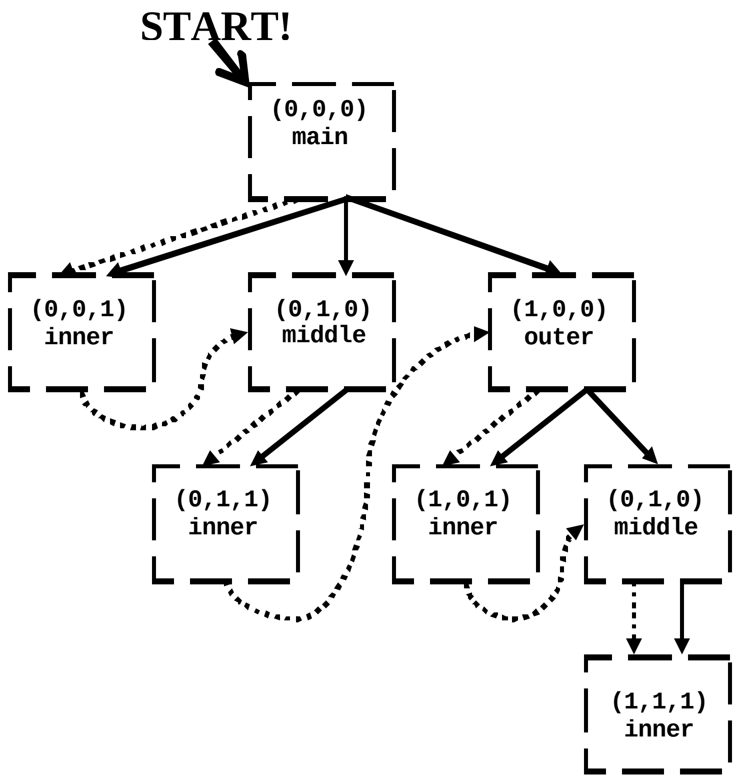


Figure 6: A graphical representation of the code in Figure 5, assuming `a`, `b`, and `c` all equal 2. The numbers in parentheses correspond to the program's loop iteration, `(i,j,k)`, while acting as indices for each node. For example, the thread on node `(1,0,0)` has its `i = 0`, `j = 0`, and `k = 0`. Each box in the figure represents a thread, and the solid lines show the hierarchy of all the spawn threads and illustrate the total amount of loop parallelism. In contrast, the dotted-lines reveal the sequential ordering of the code if it was executed serially.

After examination, it is obvious that the code for the `main` thread and the `outer` thread are very similar. The only difference is their usage of the variable `i`. Unlike the `main` thread, which initializes the variable `i`, the `outer` thread must receive the value for `i` from the thread executing the previous outer loop iteration. Consequently, the `outer` thread must perform a `test_cbit` operation to check if it has received the value of `i`. If the `test_cbit` is successful, the thread then dequeues the value from the Cqueue and stores it locally. The `middle` and `inner` threads both share the same structure with the `outer` thread. Figure 6 gives a graphical representation of the code in Figure 5, assuming `a`, `b`, and `c` all equal 2.

5.2 Exploiting Even More Parallelism with LPthreads

To further demonstrate the versatility of LPthread, we can decrease the granularity of each thread and extract even more parallelism, as shown in Figure 7. Instead of having three threads (`outer`, `middle`, and `inner`) that represent the different level of the nested loop, we now create three threads that represent the execution of each subroutine: `use1`, `use2`, and `use3`. As evident from the code in Figure 7, each thread carries only one `use` subroutine. Again, LPthread can spawn only one thread in each loop level. Aside from the thread that executes the `use3` subroutine, which is executed only in the inner loop, each thread can now spawn a maximum of two child threads.

```
main()
{
    i = 1;
    j = 0;
    if (i < a)
      thread_ptr1 = spawn(do_use1,1,0)
      thread_ptr1->Cqueue[0].enqueue(i)
      thread_ptr1->update_Cbit(0)
   thread_ptr2 = spawn(do_use2,1,0)
    thread_ptr2->Cqueue[0].enqueue(j)
    thread_ptr2->update_Cbit(0)
    use1()
    stop
}
```

```
do_use1()
{
  j = 0;
  test_Cbit
  i = Cqueue[0].dequeue
  i = i + 1
  if (i < a)
    thread_ptr1 = spawn(do_use1,1,0)
    thread_ptr1->Cqueue[0].enqueue(i)
    thread_ptr1->update_Cbit(0)
  thread_ptr2 = spawn(do_use2,1,0)
  thread_ptr2->Cqueue[0].enqueue(j)
  thread_ptr2->update_Cbit(0)
  use1()
  stop
}
```

```
do_use2()
{
  k = 0;
  test_Cbit
  j = Cqueue.dequeue
  j = j + 1
  if (j < b)
    thread_ptr1 = spawn(do_use2,1,0)
    thread_ptr1->Cqueue[0].enqueue(j)
    thread_ptr1->update_Cbit(0)
  thread_ptr2 = spawn(do_use3,1,0)
  thread_ptr2->Cqueue[0].enqueue(k)
  thread_ptr2->update_Cbit(0)
  use2()
  stop
}
```

```
do_use3()
{
  test_Cbit
  k = Cqueue[0].dequeue
  k = k + 1
  if (k < c)
    thread_ptr1 = spawn(do_use3,1,0)
    thread_ptr1->Cqueue[0].enqueue(k)
    thread_ptr1->update_Cbit(0)
  use3()
  stop
}
```

Figure 7: Using LPthread to exploit even more loop level parallelism from the code in Figure 4 by reducing the grain size of the threads.

Figure 8 gives a graphical representation of the code in Figure 7. Notice that the dash boxes surrounding each `use1`, `use2`, and `use3` thread with the same indices are the equivalents of the `outer`, `middle`, and `inner` threads in the Figure 6.

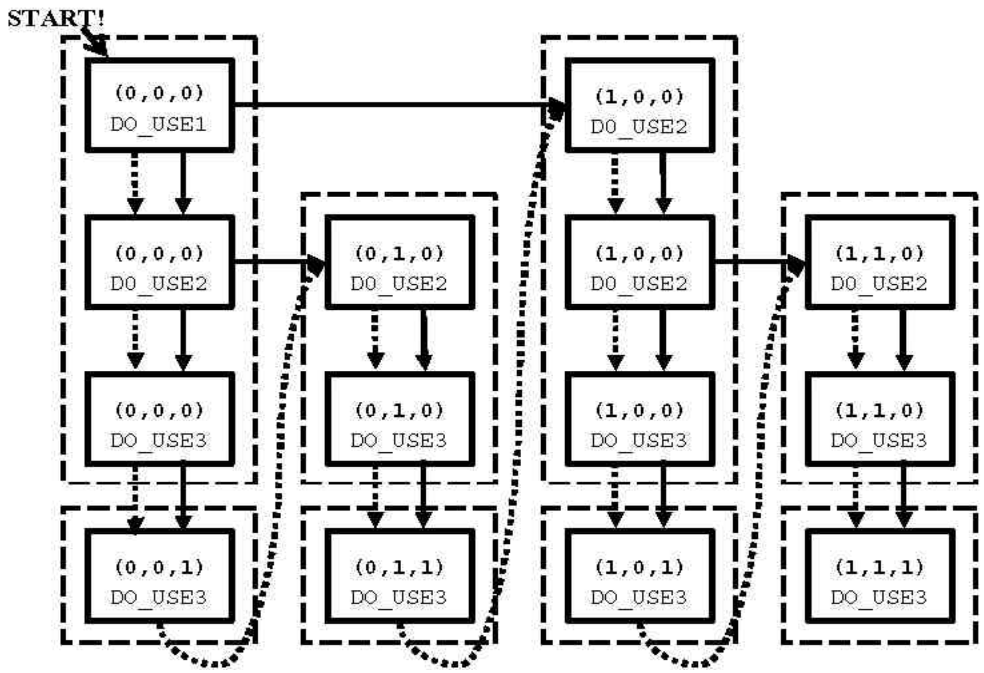


Figure 8: A graphical representation of the code in Figure 7 using the same assumptions and symbols as in Figure 6. All nodes boxed within the dash-lines correspond to the same thread with the same indices in Figure 6. More parallelism is exposed by splitting the dash-line threads into smaller threadlets.

Finally, while all the spawn threads in Figures 5 and 7 are initialized to start in the YIELD state, the code will be equally correct if they start from the WAKEUP state. For example, if the `outer` thread starts out in the WAKEUP state, it could begin executing all instructions before its `test_cbit` operation. Meanwhile, if the `outer` thread's parent managed to send the message to its child's Cqueue and updates its child's Cbit, the outer thread will successfully complete its `test_cbit` operation and continue execution. Otherwise, the `outer` thread will yield and wait for its parent. The code in Figure 7 exposes more parallelism by spawning more threads and having more `use` subroutines simultaneously in flight. On the other hand, it is also more sensitive to the cost of a thread spawn than the code in Figure 5.

6 SCHEDULER

The scheduler is by far the most important component of the LPthread and performs three critical tasks. First, the scheduler processes all the atomic operation mention in Section 4. Second, the scheduler is in charge of memory allocation for newly spawned thread as well as memory deallocation when the thread retires. Finally, the scheduler is responsible of selecting the proper threads for execution, such that the system can guarantee forward progress.

6.1 Handling Instruction Primitives

All four primitives presented in Chapter 4 communicate directly with the scheduler. Stores to the Cqueues are the only LPthread related operation that bypasses the scheduler and does directly to memory. For the LPthread model to remain deadlock-free, the scheduler must guarantee the atomicity of each operation communicated from a thread to the scheduler. The scheduler handles all requests in FIFO order and only starts handling a new request once it has completely processed the previous one.

6.2 Memory Management

While the goal of LPthread is to exploit as much thread level parallelism as possible, it must also adhere to memory resource constraint. Consequently, the LPthread scheduler monitors the total memory usage of all its spawned threads. When a processing unit requests a thread spawn, it is the scheduler's job to verify that enough memory can be allocated for the new thread. If the available memory resource is insufficient, the scheduler will yield the requesting thread. Meanwhile, the scheduler must also deallocate all threads from memory that executes the `stop` operation while examining if the newly available resource is enough to wakeup a yielded thread with it Spawn_failed bit set to 1.

6.3 Deadlock-Free Thread Selection

In general, the scheduler selects a WAKEUP thread for execution when the number of currently running threads is less than the maximum number of running threads. In addition, the scheduler will try to select a new WAKEUP thread for execution when a running thread yields or retires.

A thread model is only viable if it is deadlock-free. To guarantee forward progress, the scheduler must correctly prioritize all its threads. For example, if all threads have equal priority, the code in Figure 5 might end up heavily spawning and executing outer loop threads while the inner loop thread remains idle. Deadlock occurs when enough of the outer loop threads exist while consuming all the available memory; hence, no inner loop thread can either spawn or execute.

Since LPthread is designed to exploit thread level concurrency in serial programs through loop parallelization, any code that is transformed to utilize LPthread will still maintain the original sequential program's serial ordering. The LPthread scheduler uses this serial ordering to guarantee forward progress.

The LPthread scheduler assigned priority to the threads according their control-flow sequence within the serial program, which is made available to the scheduler through the spawn constraints in Chapter 4. In other words, in a single threaded system, a program that utilizes the LPthread model will execute in the exact same control-flow order as its sequential counterpart. To further illustrate, Figure 9 shows how the LPthread scheduler will prioritize the threads in Figure 6.

Highest Priority	(0,0,0)
│	(0,0,1)
│	(0,1,0)
│	(0,1,1)
│	(1,0,0)
│	(1,0,1)
▼	(1,1,0)
Lowest Priority	(1,1,1)

Figure 9: The thread priority of each node in Figure 6 corresponds to lexicographic ordering.

Notice that the priority scheme corresponds to the control-flow sequence of Figure 6, shown in dotted-lines. Deadlock is prevented because the thread that immediately proceeds next in the control-flow sequence is guaranteed to execute.

The scheduler implements the above priority scheme, by building a *Spawn Tree*. A Spawn Tree is a directed acyclic graph of all the spawned threads in the system, similar the ones shown in Figures 6 and 8. Every node represents a spawned thread while holding all the status bits of the thread. The node's parent corresponds to the LPthread that created the node. The node's child corresponds to all child threads that it created. In addition, a node's rightmost child points to its oldest child thread or the child thread spawned on its outermost loop. In contrast, a node's leftmost child points to its youngest child thread or the child thread spawned on it innermost loop.

The scheduler makes a thread selection by performing a depth first preorder search on its Spawn Tree for a thread in the WAKEUP state, i.e., nodes with its Ready bit set to 1 and Running bit set to 0. The scheduler immediately ends the search when a thread in the WAKEUP state is found, moved to the EXECUTE state, and is assigned to a processing unit.

To illustrate, assume that all threads in Figure 8 are in the WAKEUP state. If one was to traverse the Spawn Tree in Figure 8 while assigning the highest priority for the first node found in the WAKEUP state and assigning the lowest priority for the last node found in the WAKEUP state, a depth first search traversal would render the exact same result as shown in Figure 9.

In order to keep the scheduler efficient, the Spawn Tree should be as compact as possible to minimize search time. The LPthread scheduler compacts its Spawn Tree by retiring nodes that are no long needed. Each node tracks the number of direct descendents it has, which we call *child nodes*. A node can be retired only if the following two conditions are met:

(1) The node's corresponding LPthread have retired.

(2) The node has at most one child node.

Obviously, a node cannot retire and unlink itself from the Spawn Tree immediately after its thread finishes execution, since its children nodes will also become unlinked. A node is removed immediately from the Spawn Tree once the two conditions are met. A node without any child node is removed from the Spawn Tree as soon as its thread retires. Meanwhile, a node with only one child node must not only unlink itself from its parent node upon removal but also link its child node to its parent node.

7 DISCUSSION

In this section, the strengths and weakness of the LPthread model are discussed. This section also evaluates many of the design considerations for the LPthread model. The LPthread model is work-efficient with minimal hardware support while allowing the communications between LPthreads to have the same cost as a typical store instruction. Meanwhile, serializing the scheduler's incoming request might hinder overall system performance.

7.1 Serial/Parallel Program Interpolation

While the LPthread was mainly designed to provide a deadlock-free model for exploiting thread-level parallelism in sequential programs, LPthread is also intended to be work-efficient. Since LPthread is used to parallelize loops in a sequential program, an LPthread will create at least one child thread during its execution, depending on the depth of the loop in the sequential program. Assuming that each LPthread is independent, as the number of processing units increases, so does the number of in-flight LPthreads. For example, at steady state, the number of in-flight LPthreads = maximum depth of a loop + number of processing units. Therefore, the performance of an LPthread program with minimal interthread dependencies increases almost linearly with the number of processing units. As stated in Section 1.2, the LPthread model achieves an execution time of $T_p = O(T_1/P+T_8)$.

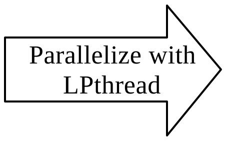
7.2 Efficient Communication and Fast Wake-Up

Ideally, every LPthread should be independent to achieve maximum thread-level parallelism; however, loop carried dependencies are common in sequential programs. Consequently, the LPthread model must handle interthread dependencies efficiently by ensuring a low overhead interthread communication and by having a fast wake-up once the dependencies are resolved.

The mechanism for LPthread communication is best explained by example. Figure 10 demonstrates how a singly nested loop can be parallelized with LPthreads. Although the serial code contains three loop carry dependencies (variables `i`, `x`, and `y`) subroutines `use1` and `use2` are data-independent. To maximize performance, the parallelized loop executes as many instances of `use1` and `use2` as possible while the loop carry dependencies `x` and `y` are being resolved.

Serial Loop

```
1 do
2  i = i + 1
3  use1()
4  use2()
5  x = use3(x)
6  y = use4(y)
7  use5(x,y)
8 while i < N
```



Parallelized Loop

```
loop()
{
  1  test_Cbit
  2  i = Cqueue[0].dequeue
  3  if (i < N)
  4    i = i + 1
  5    thread_ptr1 = spawn(loop,2,0)
  6    thread_ptr1 = Cqueue[0].enqueue(i)
  7    thread_ptr1->update_Cbit(0)
  8  use1()
  9  use2()
  10 test_Cbit
  11 x = Cqueue[1].dequeue
  12 y = Cqueue[1].dequeue
  13 x = use3(x)
  14 y = use4(y)
  15 thread_ptr1->Cqueue[1].enqueue(x)
  16 thread_ptr1->Cqueue[1].enqueue(y)
  17 thread_ptr1->update_Cbit(1)
  18 use5(x,y)
}
```

Figure 10: Parallelizing a simple singly nested loop with LPthread.

The parallelized loop iteration is divided into two execution phases, separated by its two `test_Cbit` operations. The first execution phase consists of line 1 ~ line 9, where the code receives its loop carry dependent variable `i`, computes the new `i` value for the next iteration, spawns the next loop iteration, sends the new `i` value to its child thread, and completes subroutines `use1` and `use2`. The second execution phase consists of line 10 ~ 18, where the code receives loop carry dependent variables `x` and `y`; computes the new `x` and `y` value for the next iteration by executing subroutines `use3` and `use4`, respectively; sends the new `x` and `y` value to its child thread; and completes the iteration by executing `use5`.

Although every thread communicates only with its parent, each thread utilizes two Cqueues, one for each execution phase. Once all the necessary message is sent to Cqueue[0], the thread not only executes its own instructions in the first execution phase but also starts execution of the next loop iteration in parallel. The domino process of executing a thread while spawning a new thread is continued until either the system's memory resource is depleted or the loop's stopping condition is met, e.g., `i >= N`. Meanwhile, as all the necessary messages are sent to Cqueue[1], another domino process takes place. The thread not only is woken up to complete its own instructions in the second execution phase but also resolves the next iteration's loop carry dependencies and wakes up its child thread for execution. The serial loop is parallelized by allowing simultaneous execution of `use1`, `use2`, and `use5` from different loop iterations.

Since a loop iteration is broken up into execution phases, upon completion of the thread's first execution phase, it will yield unless the dependencies for the second execution are resolved. Unlike most thread models that perform interthread communication through either atomic operations or locking mechanisms, LPthreads communicate between threads with a simple store operation. As shown in Figure 10, both `enqueue` and `dequeue` operations are non-atomic and lock free; therefore, communications between LPthreads does not hinder system performance. A thread waiting on multiple messages is only awoken once its head Cbit is explicitly set by the sender; the messages are automatically queued. Since every Cqueue can receive messages from only one sender, this feature allows the immediate wake up of an LPthread while eliminating the need for dependency counters. By assigning exclusive communication channels to each of the

LPthread's senders, communication between LPthreads has the same cost as a typical store instruction. Also, by making thread wake-up explicit across the communication channels, yielded LPthreads wake up with low overhead.

7.3 Compiler Support

Obviously, the LPthread model requires enormous compiler support for finding thread level parallelism through control dependence analysis and data dependence analysis. Although every loop with a sequential program can be transformed into an LPthread and maintain program correctness, such ad hoc approach will most likely degrade performance. Therefore, the compiler must account for the spawn and communication cost of a thread when determining whether a particular loop should be parallelized.

7.4 Hardware Support

For an LPthread program to be completely deadlock-free, the system must meet the minimum memory requirement of the program. In other words, the system must have at least enough memory resource to execute the program's *largest thread* and successfully spawn its child thread(s). We define a program's largest thread to be the thread that requires the most memory resources to execute. The memory requirement of the largest thread can be evaluated statically, since it is proportional to the largest thread's static loop level.

As mentioned in Chapter 4, the system must support four primitives for the LPthread model to be successful. It is also worthwhile to implement LPthread scheduling algorithm, Section 6.3, in hardware, since it is such an intricate part of the thread model. Figure 11, shows a logic diagram of how the Scheduler can be implemented in hardware. All requests from the processing units go through the Message Arbitrator and are pushed onto the Request Queue. The Scheduler handles the request in FIFO order from the Request Queue and replies back to a Processing Unit through the Message Arbitrator.

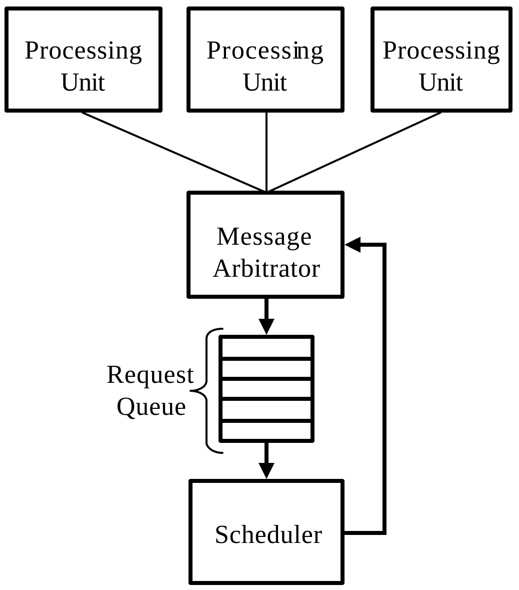


Figure 11: The relationship between the Scheduler and the Processing Units. All requests and replies are handled by the Message Arbitrator. The Request Queue is used to ensure the serialization of the incoming requests.

7.5 The Scheduling Algorithm in Hindsight

The LPthread's scheduler creates a deadlock-free environment for *N*-dimensional loop parallelization; however, the serialization of the four atomic operations could potentially become the parallel program's bottleneck. As illustrated in Figure 11, it is essential that the scheduler handles all incoming requests sequentially, since the current LPthread implementation adheres strictly to the sequential control-flow priority scheme in Section 6.3. As shown in Chapter 4, thread wake-up is performed by the `spawn` and the `update_Cbit` primitives while the thread selection is done by the `test_Cbit` and the `stop` primitives. Serialization of the incoming requests guarantees that the scheduler will choose the highest priority thread during selection. For example, the scheduler handles an `update_Cbit` operation and a `test_Cbit` operation independently in parallel. Assume that the `update_Cbit` operation wakes up a thread with a higher priority than all other threads in the WAKEUP state. The scheduler might not choose the highest priority thread when the `test_Cbit` operation fails, because it is unaware of the new woken up thread produced by the parallel `update_Cbit` operation.

Although the scheduler could potentially be optimized by processing multiple requests in parallel while performing dependency checks or employing a weaker priority scheme, the parallel program's forward progress becomes harder to guarantee. Because the scheduling algorithm could be on the program's critical path, further study is required to truly understand the latency of each primitive as well as the latency from the scheduling overhead in actual hardware/software implementation.

7.6 Memory Speculation

The LPthread model assumes that no memory conflict or memory aliasing exits between threads. It assumes an oracle memory disambiguation, where a load from one thread to the same store address of another thread would not produce a racing condition. Either the compiler must guarantee that no memory aliasing exists or a hardware mechanism is present to handle memory speculation.

7.7 Speculative Thread Spawn

As with branch prediction in serial loops, an LPthread can also be speculatively spawned to maximize performance. However, the LPthread must be squashed when the speculation is incorrect. To guarantee correctness, not only must the scheduler squash the first miss-speculated LPthread, but it must also flush all of its successors on the Spawn Tree.

8 CONCLUSION

The LPthread model was constructed as a necessary medium for exposing thread-level parallelism in *N*-dimensional do-across loops while adhering to realistic memory resource constraints. It was designed by gaining insight from the properties of control dependence, data dependence, and nested loops. As with all engineering work, the LPthread model went through several revisions in the effort to make it more robust and efficient. While the current LPthread model is deadlock-free and work-efficient, further study is required to truly understand the latency of each primitive as well as the latency from the scheduling overhead in actual hardware/software implementation. Further optimization is always possible as more insight is gained about the LPthread model. As previously stated, the LPthread model requires extensive compiler analysis to fully exploit loop parallelization and achieve speedup; therefore, a logical next step in research would be to perfect the finding of loop parallelism and code transformation.

REFERENCES

[1] J. E. Smith and G. Sohi, "The microarchitecture of superscalar processors," in *Proceedings of the IEEE*, vol. 83, pp. 1609-1624, December 1995.

[2] S. Palacharla, N. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Intl. Symp. On Comp. Arch. 24*, June 1997.

[3] M. Brown, J. Stark, and Y. Patt, "Select-free scheduling instruction scheduling logic," in *MICRO 34*, December 2001.

[4] R. P. Colwell et al., "A VLIW architecture for a trace scheduling compiler," in *ASPLOS 2*, April 1987.

[5] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai. "Code generation schema for modulo scheduled loops," in *MICRO 25*, December 1992.

[6] W. W. Hwu et al., "The superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229-248, 1993.

[7] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *22nd Int'l Symp. on Computer Architecture*, June 1995.

[8] M. Taylor et al., "The raw microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, pp. 25-35, March-April 2002.

[9] M. S. Lam and R. P. Wilson, "Limits of control flow on parallelism," in *19th Intl. Symp. on Comp. Arch.*, May 1992.

[10] E. Rotenberg, Q. Jacobson, and J. Smith, "A study of control independence in superscalar processors," in *5th Int. Symp. on High-Performance Comp. Arch.*, 1999.

[11] R. D. Blumofe, "Cilk: An efficient multithreaded runtime system," in *Proc. of the 5th ACM Symp. on Principles and Practice of Parallel Programming (PPoP)*, July 1995.

[12] G. Blelloch, P. Gibbons, and Y. Matias, "Provably efficient scheduling for languages with fine-grained parallelism," in *Proceedings of the 7th Annual ACM Symp. On Parallel Algorithms and Architectures*, July 1995, pp. 1–12.

[13] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek, "Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine," in *Proceedings of the 4th Intl. Conference on Arch. Support for Programming Languages and Operating Systems*, 1991, pp. 164-175.

[14] G. S. Sohi, S. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *22nd Intl. Symp. on Comp. Arch.*, June 1995.

[15] J.-Y. Tsai and P.-C. Yew, "The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation," in *PACT*, 1996.