Branch-Prediction in a Speculative Dataflow Processor*

Bradley C. Kuszmaul† and Dana S. Henry‡

Abstract

A processor with an explicit dataflow instruction-set architecture may be able to achieve performance comparable to a superscalar RISC processor, even on serial code. To achieve this, the dataflow processor must support speculative operation, especially speculative branches, and a pipeline with bypassing for serial code. This paper outlines a set of mechanisms to implement speculative operation with a bypassing pipeline, in a paper design called the Speculative Dataflow Processor (SDP).

The SDP uses several novel ideas as compared to traditional dataflow processors. Branches are predicted and speculated using a new branch firing rule. Several branch statements are grouped together so that they use a single branch prediction. The scheduling and bypass logic is similar to, but simpler and faster than, the corresponding logic in a superscalar RISC processor.

Speculation introduces some new compiler issues. Additional care must be taken by the compiler to prevent speculative tokens from Iteration $i+1$ from overrunning the nonspeculative tokens from Iteration i of a loop.

1 Introduction

Processors with explicit dataflow instruction-set architectures (for example [PC90, GKW85]) have generally not been as fast as contemporary von Neumann processors. They have performed especially poorly on programs that have little parallelism. One approach to solve this problem is to design processors that are a hybrid of dataflow and traditional RISC processors to obtain the best of both worlds, executing both serial and parallel code efficiently. (See for example P-RISC [NA88] and simultaneous multithreading [TEL95] at the RISC end of the spectrum and EM-5 [SKY91] and the Tera MTA [ACC+95] at the dataflow end.) This paper argues that a "pure" dataflow processor can also compete effectively if two problems are solved: It must have speculative branch execution, and the pipeline must be very efficient for serial code. We present here a paper design of a processor, called the Speculative Dataflow Processor (SDP), that we believe will work reasonably well based on our analysis and also on the intution we have gained from several compiler and processor VLSI projects. We have not yet implemented a compiler, simulator, or a circuit design for SDP, although we are working on the compiler and simulator.

Our goal is to design a dataflow processor that competes effectively with a superscalar microprocessor. This means that we are not interested in high processor utilization, for example. Contrast this approach with, for example, the Tera MTA architecture [ACC+95]), which attempts to achieve high processor utilization and is willing to use a very expensive memory system to achieve it. A processor does not need to achieve high utilization of its ALU or VLSI, since VLSI is cheap. Since the memory system dominates the cost of a high-performance machine, it would suffice to achieve high memory-subsystem utilization.

In addition to branch prediction, a dataflow processor should speculate on load/store conflicts, but there is not space here to discuss that mechanism.

Figure 1 illustrates the mechanisms needed to implement branch speculation in our dataflow architecture. (Here we are describing the state of the machine with tokens drawn on arcs, but as we shall see later, we use an explicit-token store design in which the tokens correspond to entries in an activation frame.) Figure 1(a) shows a C code segment that we compiled to the dataflow graph in Figure 1(b). The graph contains arithmetic operators, such as "+", together with switch operators (which implement branches), identified by diamonds. Switch operators have two inputs: a predicate (shown entering from the left of the diamond) and a datum (shown entering from above the diamond.)

Several switches may share the same predicate. In Figure 1(b), two switches share the same predicate, "<". To help remind the reader that the switches are related we draw the switches with the same shared predicate on the same horizontal row. In our implementation, we will take advantage of their shared predicate to reduce their speculation costs. We distinguish between switches and branches as follows: A switch can route a single token according to a predicate. A branch is the collection of switches that implement a single branch in the original program. That is, a branch is the set of switches sharing the same predicate.

Except for switches, each operator in Figure 1(b) uses the traditional dataflow firing rule—the operator fires once a value, called *token*, arrives at each input. Switch operators may fire twice, however. A switch can fire whenever its data token (vertical input) arrives. If the switch's predicate token (left input) has not yet arrived, the switch may predict the predicate's value and passes the data token to the `T` output or the `F` output accordingly. The switch fires again once the predicate token arrives. If the predicate token's value does not agree with the prediction, the switch initiates branch recovery.

To illustrate how a switch recovers from misprediction, Figure 1(b) and Figure 1(c) show the runtime state immediately before and after a misprediction. We assume that, initially, one input token was inserted along each input arch `x`, `y`, `z`, and `w`. Solid tokens within each graph indicate data that has not yet been consumed. In Figure 1(b), the comparison operators, "<" and ">2", have not yet consumed their input tokens. At the same time, however, all switches have already predicted their outcomes. The predicted outcomes are shown in bold inside each diamond. Based on these predictions, tokens have already propagated all the way to the multiply operation.

To enable recovery from misprediction, we must remem-

*This research was partially supported by NSF CAREER Grant 9702980 (Kuszmaul) and NSF CAREER Grant 9702281 (Henry.)

†Akamai Technologies and Yale University. `http://eecs.yale.edu/~bradley`

‡Yale University. `http://eecs.yale.edu/~dana`

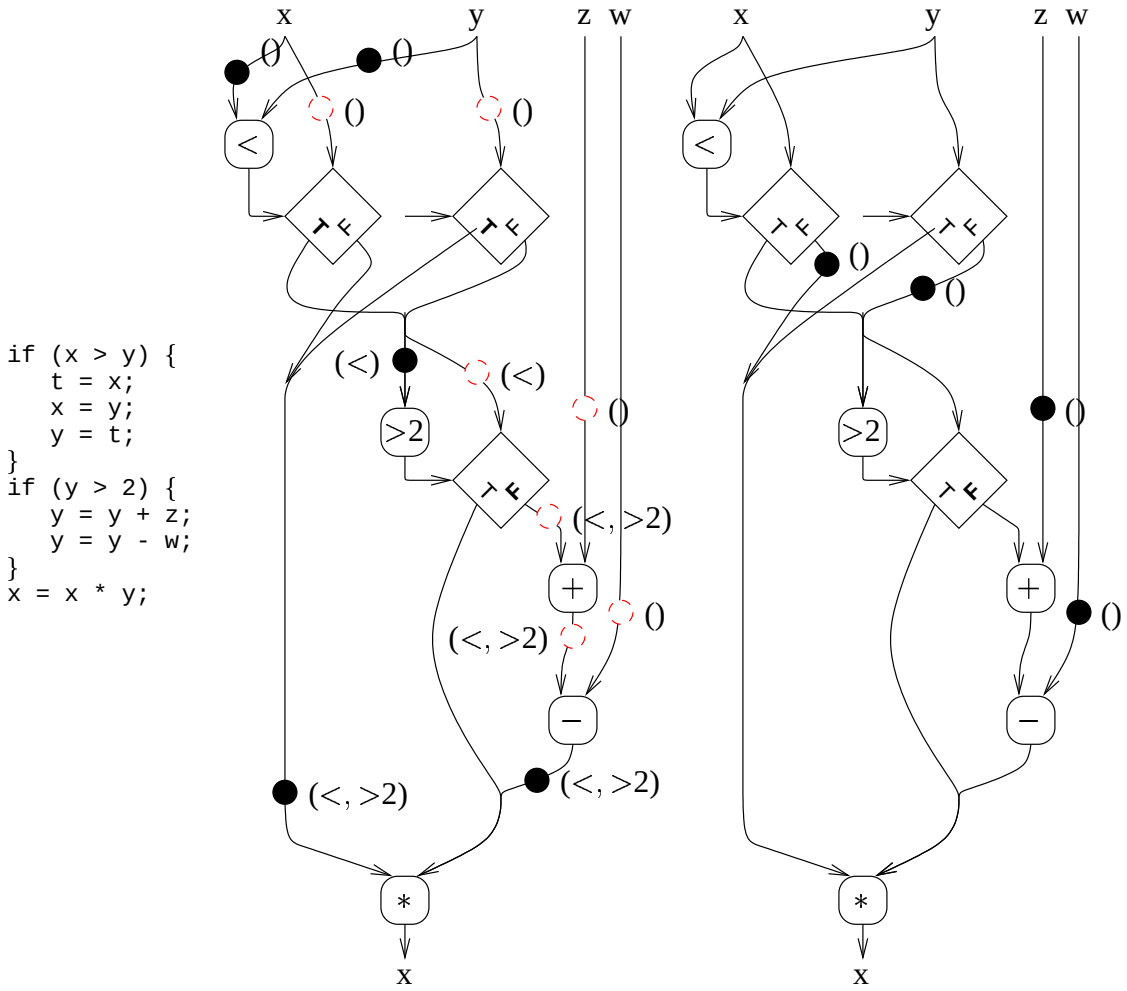


Figure 1: An example of how speculation works. (a) A fragment of C code. (b) The runtime state with the "<" test predicted true, and the ">2" test predicted false. (c) The restored graph after the "<" test resolves to false.

ber some tokens even after they have been consumed. These tokens are illustrated with dashed lines in Figure 1(b). The dashed lines show the input tokens of every operator that has fired speculatively. We keep track of which operators have fired speculatively by marking each token with the list of predicates on which it is speculating. For example, we remember the input tokens of the "+" operator because the operator's left input token is speculating on the outcome of both predicates.

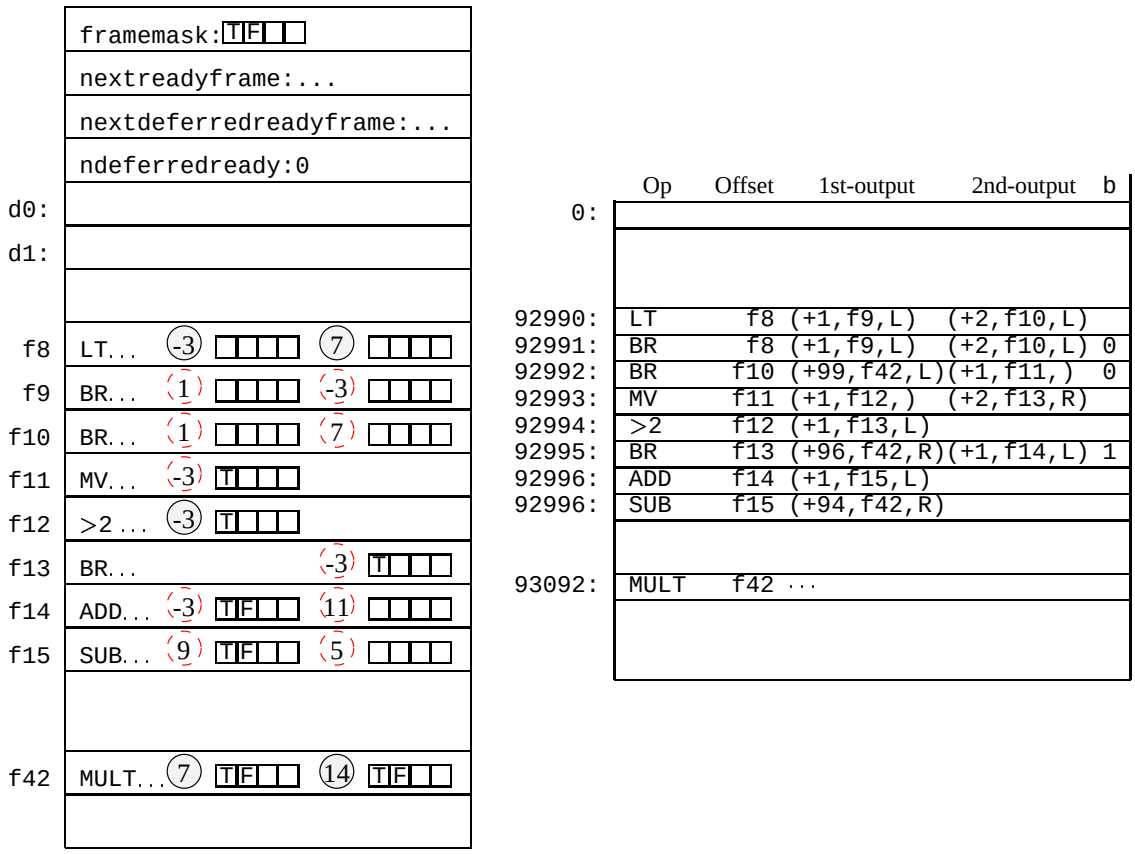
Figure 1(c) illustrates recovery from misprediction. In this example, the "<" operator has resolved to False; the affected switches have fired again and detected a misprediction. As a result, every operator that speculated on the predicate "<" undoes its computation, restoring any input token that it should not have consumed.

The rest of this paper is organized as follows. Section 2 describes the SDP instruction-set architecture. Section 3 sketches how to implement branch speculation in SDP. Section 4 argues that SDP should compare well to a superscalar processor. Section 5 discusses compilation issues raised by speculation. Section 6 shows how to support provably efficient multithreaded scheduling, and Section 7 concludes with a discussion of related and future work.

2 Instruction Architecture

Having outlined the idea behind the mechanism in Section 1, in this section we describe the instruction set architecture (ISA) for the SDP. The rest of the paper will then describe the implementation issues for this ISA.

Except for switches, the SDP processor's instruction set architecture is analogous to the explicit token store architecture pioneered by Monsoon [PC90]. Figure 2 illustrates the architected state using the code segment and execution graph from Figure 1. The state consists of set of frames, such as the one shown in Figure 2(a), and instruction memory The instruction memory holds the static information about the program (the "text" of the program), whereas the frames hold dynamic information for the procedure's outstanding instructions. Each frame corresponds to one procedure invocation or one thread,



(a) One frame (of many.)

(b) Instruction Memory.

Figure 2: Architected State.

typically.

2.1 The Frame

A frame is a contiguous region of memory which is used as the backing store for state that is normally kept in the processor core. When there are many active frames, the processor will need to move some of the frame state out of the core to the memory.

The frame includes

- a `framemask` which is used in branch speculation,
- a collection of frame entries (`f0`, `f1`, $ldots$),
- fields to implement ready-to-execute instructions, and
- fields to implement another set of deferred instructions.

Each of these are described below.

The Frame Mask

The framemask keeps track of all of the outstanding unresolved branches for a frame. Switches that use the same predicate share one branch-mask entry. In Figure 2, the first entry in the mask lists the prediction made by the "<" predictor in Figure 1; the second entry in the mask lists the prediction made by the "> 2" predictor in Figure 1.

The frame mask is part of the architected state because the compiler must manage the allocation of the frame mask entries.

The Frame Entries

For each instruction that has a token on one of its inputs, the frame keeps track of the instructions arguments and state. In Figure 2(a), arguments that have not yet been consumed appear inside a shaded token, and arguments that have been speculatively consumed appear inside a dashed token. There is an additional *argument mask* stored with each argument token which is part of our impleentation and will be described in the following section.

The Ready-to-Execute Set

Each frame keeps in its state the set of all instructions that are ready fire. More than one frame may have instructions which are ready to execute, however. The frame provides storage, called `nextreadyframe`, to build a linked list of all such frames.

The Deferred Set

Another set of instructions, called the deferred set, is also kept by the system. In the frame the `nextdeferredreadyframe`, `ndeferredready`, and `d`i locations store a per-frame list of deferred instrutions. This deferred set supports a provably efficient scheduler for multithreaded programs, and its rationale and behavior is described below in Section 6.

2.2 The Instruction Memory

The assembly format of an arithmetic instruction consists of:

`address:` `opcode` f (i_1, f_1, p_1) (i_2, f_2, p_2)

where `address` is the instruction's address in instruction memory, f is an index into the frame, i is an offset in instruction memory starting from the current instruction, and p is an instruction's input port (Right or Left.) The individual instruction fields are

`opcode`: the operation,

f: the index of the instruction's frame entry,

`address` $+ i_1$: the address of the first output's instruction,

f_1: the index of the first output frame entry,

p_1: the input port of the first output,

and similarly for the second output's address, index, and port.

In addition, switch instructions name an entry, b, in the branch mask that holds their prediction while they speculate:

`address:` `BR` f (i_1, f_1, p_1) (i_2, f_2, p_2) b.

Branches that share the same predictor share the same mask entry. In addition, static switches that never dynamically coexist within the frame may also name the same mask entry.

3 Implementing Branch Speculation

Section 2 described the SDP instruction set architecture (ISA), which is the programmer-visible behavior of the machine. This section sketches an implementation, and Section 4 argues that the implementation should be at least as fast as a superscalar pipeline.

To implement branch speculation, we added a n-bit frame mask register to the frame. The register uses 2-bits to encode the state of each entry in the frame mask. There are three states, which we notate as

$\square$: the entry is not in use,
$\boxed{\text{T}}$: the entry's predicate is predicted taken, and
$\boxed{\text{F}}$: the entry's predicate is predicted not-taken.

Each entry's value is set the first time a switch fires speculating on the entry's predicate. Each entry's value is cleared whenever a switch fires for the second time, confirming or refuting that prediction.

We also maintain a n-bit *argument mask* with each argument field in the frame. Each argument mask lists a subset of the frame mask on which the corresponding argument is speculating. Figure 2 shows the setting of all the argument masks for the program state described in Figure 1(b). For example, the `ADD` instruction's left argument is speculating on both predicates from Figure 1 while its right argument is not speculating on either.

A dedicated n-bit broadcast bus ties the frame mask to the argument masks. Whenever a predicate resolves, the bus communicates the resolved value to each argument mask. If the predicate was correctly predicted, each dependent argument simply clears the predicate's entry in its mask since it is no longer speculating on that predicate. If the predicate was incorrectly predicted, each dependent argument deletes itself and possibly reinstates its sibling to implement branch recovery.

We considered using a scheme in which mispredicted branches create "kill tokens" that follow the paths of the original speculated tokens, but we were concerned that the kill tokens might not catch up in time to avoid certain race conditions. In fact, under some conditions the kill tokens might never catch up with the tokens that they are trying to kill.

4 Performance: SDP vs. Superscalar

Now that we have discussed the implementation of branch speculation in the SDP, in this section we argue that the SDP pipeline should be as fast as a superscalar pipeline. In Section 5 we will discuss compilation issues. In this section, we describe briefly the rest of the SDP core and argue that the SDP circuitry is no more difficult to implement than a standard superscalar processor's circuitry with some parts of the circuitry simpler and faster than superscalar's. The key observation is that each entry in the SDP's frame corresponds is a superset of an entry in the superscalar's reordering buffer.

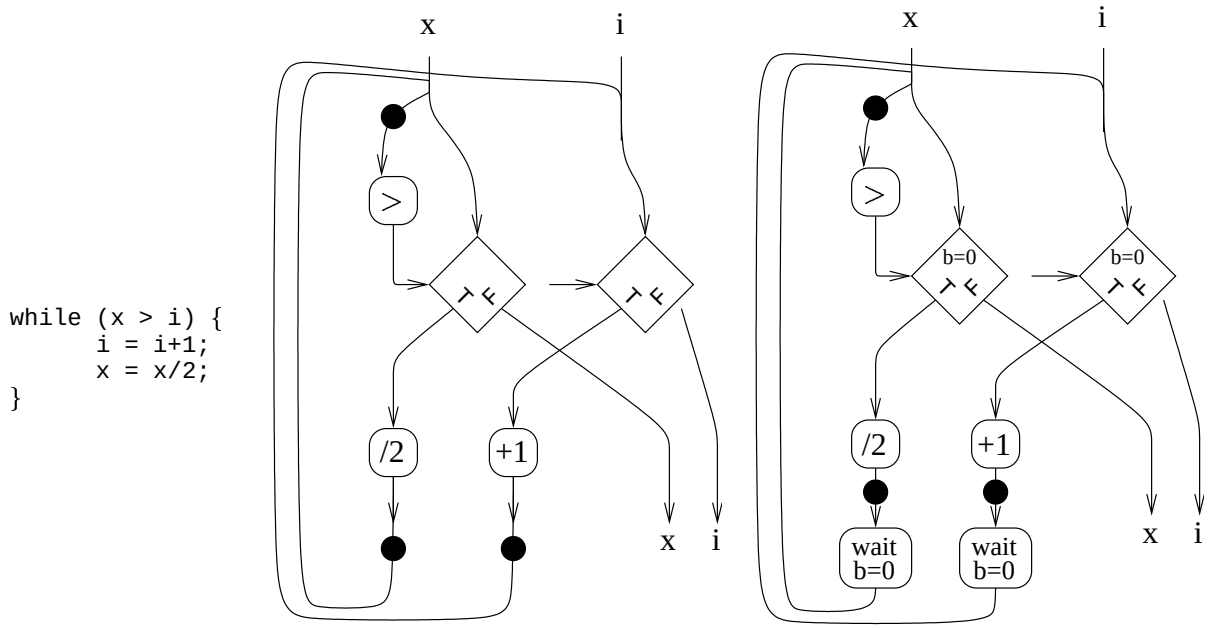
Unlike a superscalar processor, the SDP explicitly names the children of each instruction in the frame. As a result, the SDP does not have to broadcast each result to the entire frame. Instead, it can directly write each result into each child's frame entry. This optimization replaces area-intensive associative writes into the superscalar reordering buffer with faster and smaller direct writes into the SDP's frame.

However, explicitly naming each instruction's children also has its costs. If there are many destinations for an instruction, and the instruction has limited fan-out, then extra fanout instructions will be needed. In our architecture, we used instructions with fanout of two, but it may make sense to use instructions with a fanout of three or four to reduce the need for extra fanout instructions.

Also appearing in a frame entry but not in a reordering buffer entry is the argument mask described in the previous section. This mask supports selective recovery from misprediction in the SDP. Unlike a traditional superscalar processor, the SDP can back out of exactly those instructions that depend on a mispredicted branch. In contrast, a superscalar undoes *all* instructions following a mispredicted branch, whether they actually depend on the mispredicted branch or not.

The SDP does not need renaming logic since the compiler explicitly manages the reuse of frame entries. Explicitly managing storage reuse puts pressure on the size of the frame, however. It remains to be seen how large a frame is needed to achieve good performance.

The critical-path length of a program may be longer using SDP than using a serial instruction set, because in a superscalar, correctly predicted branches do not appear in the critical path of the program at all. In the SDP, even correctly predicted branches add the cost of the switch instruction to the critical path. The number of instructions can be greater in SDP than in superscalar processors as well, since a single branch in



(a) C code. (b) Compiled for traditional firing rule. (c) Compiled for speculation.

Figure 3: Loop barrier example. This code computes `i+=lg(x);x=1`.

a superscalar may correspond to many switch instructions in SDP.

In other aspects, the SDP is essentially identical to a superscalar reordering buffer entry, and executes in the same way. For example, the same bypassing techniques used by the superscalar processor can be used in the SDP.

5 Compiler Support

This section discusses compiler issues for SDP, which are important even for serial programs. The next section will discuss the hardware support needed for highly concurrent multithreaded programs. In addition to the hardware issues described in Sections 3 and 4, the compiler needs to take extra care when compiling for a speculative dataflow processor.

First, as we saw in Figure 1, the compiler must group together switches that use the same predicate. Such grouping reduces the number of outstanding predictions to the number of unresolved predicates rather than unresolved switches. In addition, the compiler must understand our new firing rule for switches. Without the compiler's cooperation, the speculative firing of a switch could yield multiple tokens along one arch in violation of our explicit-token-store dataflow architecture.

Figure 3 illustrates the effect of the new switch firing rule. It shows a simple serial C code loop compiled with the traditional single-firing rule (Figure 3(b)) and with our new speculative firing rule for switches (Figure 3(c)). In Figure 3(b) the compiler has used the traditional rule, assuming that each switch will fire only once, after both inputs have arrived. Under this assumption, all initial inputs to the loop will be consumed before the next iteration's inputs are generated. If the switches were to fire speculatively instead, without waiting for their predicate tokens, the program would fail. As Figure 3(b) illustrates, the next speculative value of `x` could reach the predicate operator "`>`" before the first value has been consumed.

To avoid multiple tokens along the input arch to the predicate operator, the compiler must introduce explicit *speculation barrier* instructions as in Figure 3(c). We have shown the branches with branch masks ("`b=0`"), and the speculatiion barrier is denoted by "`wait b=0`." A speculation barrier will not fire until the branch mask mentioned has resolved.

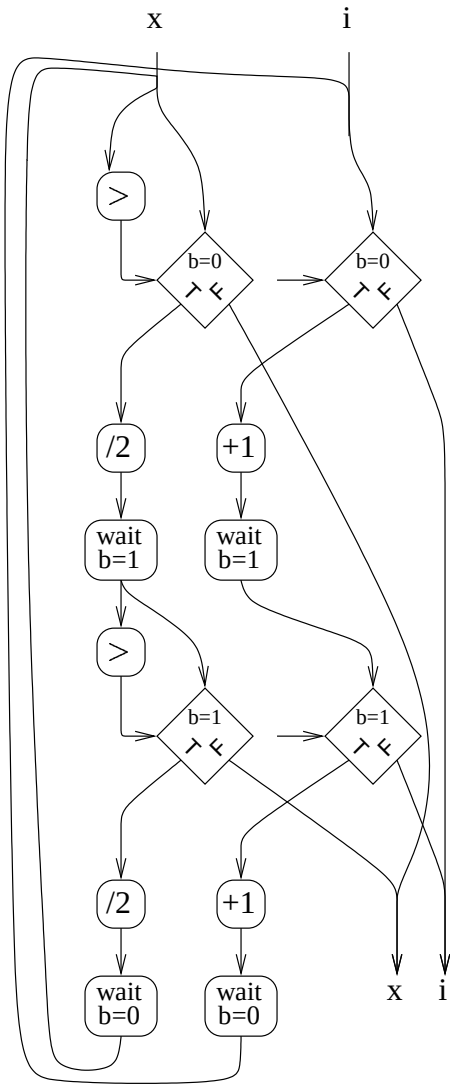
One optimization for this kind of code would be to unroll the loop. Figure 4(a) shows the code unrolled once by hand, and Figure 4(b) shows the resulting code. Note that the first set of wait instructions waits on the second branch to resolve, and the second set of wait instructions waits on the first branch to resolve. (Initially both branches start in a resolved state, which gets the loop started.) This means that the first iteration and the second (of the original loop) can execute concurrently. And then when the first iteration finishes, the third iteration can start and run concurrently with the second. Then when the second iteration finishes, the fourth iteration can start, running concurrently with the third. Thus, if the compiler unrolls k iterations of the loop, every contiguous sequence of k iterations will be able to run concurrently, even if they do not align with the unrolling.

6 Support for Parallel Programs

So far we have explained how to run serial programs on a speculative dataflow processor, taking advantage of the parallelism within one subroutine of an otherwise serial program. This section outlines how a dataflow processor can be designed to support provably efficient scheduling of highly concurrent multithreaded programs. Section 7 will then discuss the related and future work.

```
while (x > i) {
      i = i+1;
      x = x/2;
      if (!(x > i)) break;
      i = i+1;
      x = x/2;
}
```

(a) C code unrolled once.



(b) Compiled for speculation

Figure 4: An unrolled version of the code from Figure 3. When we unroll the loop we can use split-phase speculation barriers so that the two iterations of the loop can run concurrently..

The SDP can support highly concurrent programs by executing several frames concurrently. One of the problems with such programs is that if the call tree is expanded breadth-first or randomly, then the system can run out of memory easily. Many Monsoon programs had this difficulty: either they would run too slowly because they lacked parallelism, or they had plenty of parallelism but needed huge amounts of memory, and it was very difficult to tweak the program to get it to run "just right."

Our approach is to provide support for a provably efficient scheduler, such as the one used in Cilk [BJK+95]. To be concrete, we will discuss the support needed for the Cilk scheduler.

The trick is to prevent the system from allocating new frames when there are already enough frames to keep the processor busy. Figure 5 shows an example of this idea at work. A Cilk program that spawns a total of eight children (the root node spawns two children, each of which spawn to grandchildren, each of which spawn two great grandchildren) could require up to 15 frames to run if the frame allocation is not constrained. A better situation is shown We in Figure 5(c), in which part of the tree has been completed, and part of the tree is waiting to be spawned, and part of the tree is begin worked on. The part of the tree that is being worked on has at most 2 active leaves in this case, and the part of the tree that is waiting to be spawned is deferred.

To be more specific about the allocation rule, we provide here a brief review of the Cilk system, from the perspective of a multithreaded processor architect. In Cilk, the computation is structured into a call tree, in which a vertex corresponds to a subroutine instance, and in which certain subtrees can execute in parallel. To execute several subtrees in parallel, the programmer writes a collection of "spawn" procedure calls, and then a "sync" operation that waits for all the children to complete. An ordinary procedure call is simply a spawn of a single subtree, followed by a sync.

Cilk achieves optimal time and space bounds simultaneously. The time bounds are expressed using the time to execute on one processor, T_1, and the critical path length of the program, T_∞, which is the time it would take to run on an infinite number of processors. On P processors, Cilk can run a program in time that is $T_1/P + O(T_\infty)$. If the space bound on one processor is S_1, then the space bound on P processors is $P \cdot S_1$. These bounds are optimal under certain assumptions.

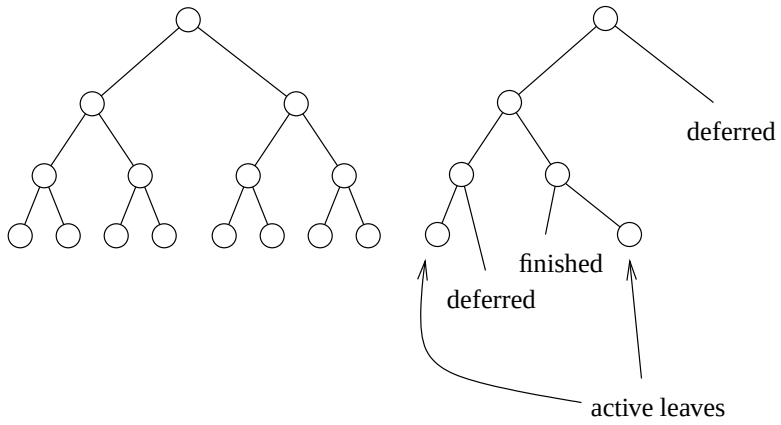
Cilk programs must be *strict* in order for the scheduler to achieve these bounds. Informally, a strict program is one in which, once a subtree starts, it is able to finish without waiting for other subtrees to finish.

Cilk achieves these time and space bounds by guaranteeing that at most P "leaves" of the call tree exist at any given time. Another way to say this is that in the tree, at most P forks are expanded at any gien time.

When the system has fewer than P leaves running, every spawn actually starts up a new subtree in parallel. When the system has P leaves active, then no new subtrees are spawned. That is, the system runs in a serial, depth-first, order on each of the extant branches of the tree. That means that only one spawned child of a frame is actually started at a time. The others wait until the first one completes, and then another spawned child can run.

For the discussion here, we are interested in supporting Cilk on a single speculative dataflow processor that may have a limited number of frames. So P, instead of referring to the number of processors, refers to the number of leaf frames we can support in the processor core. The speedup bounds work

```
int recurse (int n) {
  if (n==0) serially_work();
  else {
    spawn recurse(n-1);
    spawn recurse(n-1);
    sync;
  }
}
```



(a) The program.

(b) The entire call tree for `recurse(3);`

(c) Only two leaves are allowed to exist at a time.

Figure 5: In Cilk, a limited number of forks in the dynamic call tree are allowed. Each node of the call tree shown in (b) represents one invocation of the procedure named `recurse`. The whole call tree includes eight leaf nodes, but if on a two-processor system we only want there to be two leaves enabled. Parts of the tree have already finished executing, and so their frame memory is deallocated, and part of the tree is waiting to execute, but we do not actually allocate memory until one of the leaves finishes.

out differently as well, since there are not actually P ALUs and other computational units, but the system still has a sound theoretical basis.

Thus, to make this dataflow-oriented Cilk work requires that the runtime system be able to distinguish between two cases when spawning a child. The "serial case" is when a child is being spawned and there are no other children currently in existance. The "parallel case" is when there is already a child running for a particular frame, and we must be careful not to start another child unless there are idle processing resources.

Most of the support for Cilk-scheduling within the SDP can be implemented in software, with a very small amount of hardware support. The system must maintain a separate "deferred" execution queue for the instructions that allocate new frames. Instructions are executed via the regular window-scheduling mechanism whenever possible. The rule for when a deferred frame allocation instruction can run is more complex, however.

The idea is that frame allocation instructions in the deferred queue should not be run if there are too many spawned children in the system. To make this work the processor keeps a global count of how many leaf children are running. The processor might be designed to allow, say, 16 concurrent leaves in the call tree to be executing. If the global count is less than 16, then the processor executes a frame allocation instruction out of the deferred queue (putting the resulting tokens into the regular execution pipeline) and increments the global counter. If the global count is greater than 16, then the processor does not execute instructions from the deferred queue.

Here is how the system can compute how many active leaves exist in the call tree. In software, a Cilk program sets up a counter in the activation frame to keep track of how many children are running (the "active-child count.") Initially the counter is set to zero. When spawning a child subroutine, if the active-child count is zero, the spawn is treated like a serial call (the frae allocation instruction is executed normally), and the local counter is incremented. If the active-child count is positive, then the token that starts the frame allocation is placed into the deferred queue, and the count is not incremented.

When a forked child completes, the system must decrement the parent frame's counter, and if that goes from two to one, it must decrement the global leaf counter, which will then allow some deferred instruction (if there is one) to run, allocating a new frame.

The effect of all this is to implement a provably efficient scheduler by providing the mechanisms needed to prove the Cilk results for SDP.

We could have taken the decision to perform Cilk-style scheduling in software, but we wanted to be able to write Cilk-style programs in which the spawn and procedure call instruction-sequence are the same. We wanted the "serial case" to run as fast as possible, and so we provide hardware support for the Cilk-style scheduling.

7 Related and Future Work

The biggest difference between our machine and previous pure dataflow machines, such as Monsoon [PC90] and the Manchester dataflow machine [GKW85] is that we make extensive use of speculation to achieve high performance on single-threaded code. In contrast Monsoon could only use one eighth of a single processor's cycles on single threaded code. The speculation we propose is possible because of advances in VLSI technology since the previous generation of pure dataflow machines.

Among the hybrids, the two machines that look the most like our proposed machine are the Tera (now known as Cray) MTA, and the EM-5. Beyond the fact that our machine is a pure dataflow machine, and makes extensive use of speculation their are some other interesting differences.

The Tera MTA [ACC+95] allows very restricted out-of-order execution within a single thread (called a stream). Each instruction specifies how many successive instructions can be issued before this instruction completes, and this is limited by 7. So in effect, a single stream has a window size of 8 or less. The pipeline depth for the MTA is about 70 clock ticks, and so at least 9 streams are required to achieve 100% processor

utilization. The MTA relies on the compiler detecting many streams of parallel instructions (of the order of a few tens per processor) to get high throughput. In contrast, our approach is to implement bypasses so that, a dependant instruction can run on the next cycle immediately after the completion of its predecessor. Even so, our processor would require high parallelism to achieve 100% utilization because some operations, such as memory, take a long time, and that would require a highly parallel memory subsystem, which seems infeasible for a microprocessor using today's memory technology.

In the EM-5 [SKY91], the scheduling unit is a "strongly-connected component", which may be one or more instructions. The instructions within a component are executed in sequence. Instructions in the same strongly-connected component can be run in successive cycles, but dependent instructions in different components cannot.

We are currently building a C compiler and simulator for the SDP. Several other researchers have shown that it is possible to systematically compile serial programs for dataflow machines [BP89, NHSB94, WA95]. Given that it is possible to compile serial programs, our compiler work is directed to supporting the thesis that a pure dataflow processor can compete with a von Neumann processor. We hope to soon have results about the effectiveness of our branch prediction scheme and of our fetch prediction scheme, and of the SDP in general.

As a possible improvement to the ideas of the SDP, we are considering a dataflow processor with a very different approach to managing data and tokens. Instead of using tokens that carry data, we are considering a dataflow processor that uses explicit registers, and in which the tokens carry only synchronization information. This would reduce the number of switches to be comparable to a superscalar processor. Instead of one switch for every data value, it would be more like one switch per branch. That would allow us to remove the branch masks from the ISA because the branch masks can be dynamically assigned to the instructions. This approach would also reduce the number of instructions in the code.

Acknowledgments

Yale graduate student Rahul Sami helped with the analysis of the related work, and has been examining the problem of compiling C for a dataflow machine.

References

[ACC+95] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera Computer System. `ftp://www.net-serve.com/tera/arch.ps.gz`, 1995.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pages 207–216, Santa Barbara, California, July 1995. (`http://theory.lcs.mit.edu/pub/cilk/PPoPP95.ps.Z`).

[BP89] Micah Beck and Keshav Pingali. From control flow to dataflow. Technical Report TR 89-1050, Department of Computer Science, Cornell University, Ithaca, NY 14853-7501, October 1989.

[GKW85] J. R. Gurd, C. C. Kirkham, and I. Watson. The manchester prototype dataflow computer. *Communications of the CACM*, 28(1):34–52, January 1985.

[NA88] Rishiyur S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? CSG Memo 292, MIT Laboratory for Computer Science, November 1988. See [NA89].

[NA89] Rishiyur S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *The 16th Annual International Symposium on Computer Architecture*, pages 262–272, Jerusalem, Israel, May 1989. ACM SIGARCH Computer Architecture News, Volume 17, Number 3, June 1989.

[NHSB94] Mark H. Nodine, James E. Hicks, Cotton Seed, and Michael J. Beckerle. Generating parallelism profiles from C programs. Technical Report MCRC-TR-43, Motorola Cambridge Research Center, One Kendall Square, Building 200; Cambridge, MA 02139, September 1994. (Available as `http://csg-www.lcs.mit.edu:8001/mcrctr/tr43/ppg.html`).

[PC90] Gregory M. Papadopoulos and David E. Culler. Monsoon: An explicit token store architecture. In *Proc. 17th. Intl. Symp. on Computer Architecture, Seattle, WA*, May 1990.

[SKY91] Shuichi Sakai, Yuetsu Kodama, and Yoshinori Yamaguchi. Architectural design of a parallel supercomputer em-5. In *Proc. Japan Soc. Parallel Proc., Kobe Japan*, pages 149–156, May 14–16 1991.

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pages 392–403, Santa Margherita Ligure, Italy, 22–24 June 1995. *Computer Architecture News*, 23(2), May 1994.

[WA95] S. F. Wail and D. Abramson. Can dataflow machines be programmed with an imperative language. In G. Gao, L. Bic, and J.-L. Gaudiot, editors, *Advanced Topics in Dataflow Computing and Multithreading*, pages 229–265. IEEE Computer Society Press, 1995.