University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

A Superstrand Architecture

Andrés Márquez
Kevin B. Theobald
Xinan Tang
Guang R. Gao

CAPSL Technical Memo 14
December 1, 1997

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

In this paper, we present the *superstrand* architecture and it's underlying execution model. A superstrand architecture exploits the notion of a *strand* – a block of instructions grouped together by a compiler to become a scheduling quantum of execution — a strand is enabled at runtime if all necessary dependence (data and control) constraints are satisfied. The code is partitioned into strands so that (1) source and destination of a "long latency" data dependence will be placed into different strands; (2) the source (e.g. a test) and destinations of a branch operation will be placed into different strands. We show that partitioning strands in this way gives the hardware access to a pool of strands resulting in a large "window" of instructions, which is necessary for sufficient instruction level parallelism, but reduces the number of potential dependences which the hardware must check, speculate and resolve at runtime, leading to significant savings in hardware complexity.

The main results of this paper include:

- The superstrand program execution model is proposed and presented. An unique feature is: under this model, a strand, once scheduled for execution, can proceed to completion without stalls due to latency for branching or data access and synchronization. (This feature simplifies the superstrand architecture design — see the non-preemptivity feature below.)
- We demonstrate that programs can be partitioned into strands to be executed under the superstrand execution model with a reasonable programming effort and compiler optimization.
- We present an outline for an architecture model and support for the superstrand architecture model. One unique feature of this architecture model is that it makes use of "explicit" dependence information in the inter-strand synchronization and scheduling lookahead, which is in contrast to register scoreboarding where implicit dependence information in the sequential code needs to be discovered and resolved at runtime.

Contents

List of Figures

List of Tables

1 Introduction

In an effort to squeeze more performance out of a single CPU processor, designers are trying to increase the number of instructions which can be issued per clock cycle – so-called *instruction-level parallelism* (ILP). The dominant trend today is to make aggressive use of superscalar techniques (branch prediction and speculative execution with data/control dependence and value speculation, out-of-order execution with register renaming, etc.) in order to create a large window or pool of eligible instructions from which to choose those to issue in parallel. Yet the performance gains so far have been only modest. There is an open debate today whether such techniques are capable of achieving significantly higher issue rates (say, beyond 4–8 instructions per cycle), and, even if they can, whether the hardware required to achieve this could be put to better use elsewhere.

We believe that a suitably-designed multithreaded program execution model can serve as the basis for a multiple-issue processor; that it is possible for a modern compiler, using feasible analysis tools, to convert ordinary programs into code threaded for this processor; and that this processor can run this code with reasonable levels of parallelism, without the need for expensive hardware mechanisms to support the aggressive superscalar techniques mentioned above. Furthermore, the same processor can serve as the building block of high-performance multiprocessor systems.

In this paper, we present the *superstrand* program execution and architecture model. A superstrand architecture exploits the notion of a *strand* – a block of instructions grouped together by the user (e.g. a compiler) to become a scheduling quantum of execution. A strand is enabled at runtime if all necessary dependence (data and control) constraints are satisfied. A modern compiler, using feasible analysis tools, can convert a program into stranded code such that (1) source and destination operations of a "long latency" data dependence will be placed into different strands; (2) the source (e.g. a test) and destination operations of a branch will also be placed into different strands. As a result, the superstrand architecture has the following features:

- All instructions within a strand are unconditionally executed (i.e., there are no conditional branches within a strand).
- All intra-strand data dependences are "local" (i.e., values are passed between instructions in the same strand solely through registers).

Under a superstrand architecture model, a CPU accesses a large "window" of instructions from several concurrent strands but reduces the number of potential dependences which the hardware must check at runtime, leading to significant savings in hardware complexity. This is because of the fact that the code compiled by our compiler will retain key dependence information between strands, and the superstrand architecture will make explicit use of them in inter-strand synchronization and scheduling.

A novel hardware strand synchronization unit is introduced to implement the fine-grain dependence based synchronization between strands. Efficiency is achieved by utilizing the explicit dependence information discovered by the compiler in the following way. First, the compiler can analyze the “lifetime” of synchronization resources needed for strand enabling and allocate them efficiently to “synchronization bit registers” (called SRBs) in a small hardware strand synchronization and scheduling unit using a principle inspired by register allocation. Our solution is in contrast to a normal register scoreboarding where implicit dependence information in the sequential code needs to be discovered and resolved at runtime.

The compiler can explicitly encode the immediate dependence successor information of each strand into the generated code so the dependence does not need to be rediscovered at runtime to exploit parallelism.

2 Design Space

In this section, we review both conventional superscalar architectures and multithreaded systems. We formulate the problem of maximizing ILP as one of scheduling instructions from a window or windows, and show how the multithreading approach can lead to a simpler hardware implementation.

2.1 Design Space for Instruction-Level Parallelism

The dominant trend in the design of modern processor architectures is to use increasingly powerful hardware to detect parallelism dynamically while maintaining the sequential semantics of the program execution model. Such hardware looks at a window of instructions following the last instruction whose execution has been “committed,” and tries to find instructions which have no dependences with instructions not yet completed. The size of the window is bound by the hardware limits, and its extent is also limited by program factors such as conditional branches.

The number of instructions which can be in this window, and the number which can ultimately be issued in a given cycle, depend on the techniques used in the processor hardware. Figure 1 illustrates how the situation might appear at one point in time. Different shades represent instructions which are exposed through different ILP techniques. The following techniques are used in most modern processors or have been proposed:

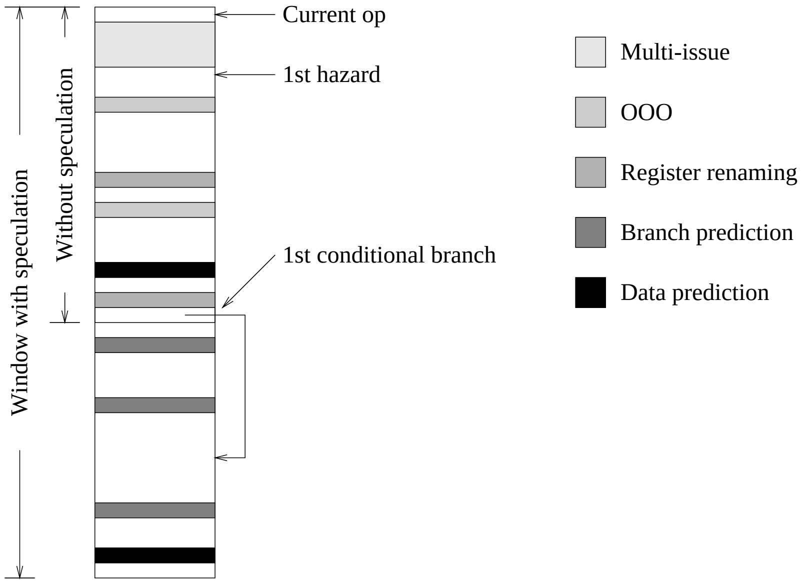


Figure 1: Multi-issue lookahead window and parallelism

- *register scoreboarding* techniques are used to detect dependences at runtime and support limited *in-order multi-issue* capabilities. The term "in-order" refers to the fact that *anti-dependence* and *output-dependence* between instructions may cause stalls at the execution or issue stages [29].

- *Out-of-order* (OOO) execution allows multiple instructions to be issued and executed out-of-order from the window, even in the case of anti- and output- dependence between them. This is accomplished with the help of register renaming, and further exploited through branch prediction and speculation as described below [14].

- *Register renaming* provides a mechanism that enables anti-dependent instructions to run concurrently without changing the sequential semantics. This allows more instructions to be selected from the window [14].

- *Branch prediction* enables the window to grow beyond the first conditional branch after the current instruction, exposing still more instructions to parallel execution. These new instructions, unlike those exposed by OOO and register renaming, are executed speculatively and squashed if the branch is predicted incorrectly [16].

- *Data prediction and speculation* (of several types) allow a more fine-grained speculation based on individual data items rather than basic blocks (as with branch prediction) [18].

It is still an open question whether these techniques alone will be enough to achieve sustained IPC levels of 4–8. Studies based on sequential program traces have shown that a large window is required for significant parallelism [3, 30]. Unfortunately, a large window means there are many potential interactions between instructions to compute dynamically, leading to an explosion in hardware complexity at least the square of the window size). Even today's processors, with only modest levels of ILP (typically less than 2 IPC for real applications), devote large regions of silicon to this task. Can higher IPC rates be achieved while keeping the complexity of the implementation under control so as not to impede high clock rates?

Dynamic reordering and speculation hardware also complicates matters for compiler optimization.

- It is not yet clear how best to utilize OOO dynamic scheduling mechanisms and register renaming at compile time. (Some people have tried [17, 24], but so far have had only very limited success.
- It is not yet clear how a compiler can make the best use of some of the proposed hardware branch prediction and speculative execution mechanisms in code optimization. For example, the two-level branch prediction scheme is expected to be widely used in future microprocessors, yet we know of no compiler which models the behavior of such a system in its code analysis. For less predictable branches, this brings the effectiveness of compile-time global code scheduling into question.
- In terms of architectural features for data speculation (e.g., data dependence speculation, value prediction, etc.), more evidence is needed to judge the effectiveness of compiler optimizations for their use. So far, this remains a difficult and challenging task.

It is also not clear to us that this is the best way to use silicon resources if much of the computation and its resource requirements are less schedulable at compile time. Furthermore, unsuccessful speculation will lead to wasted computations along the wrong predicted path.

2.2 Multithreading

Multithreading has been proposed as a competing approach to exploiting parallelism in processor architectures [6, 13]. A multithreaded computer has multiple "threads" of computation which can run concurrently. Dividing a computation into threads – in effect, having multiple program counters – increases the number of instructions per cycle in two ways. First, a multithreaded architecture can avoid stalls due to a long-latency operation, such as a remote load, by quickly switching to another task when it encounters such an operation. Second, multiple threads are a direct source of parallelism because they can be executed in parallel by separate execution units.

2.2.1 Multithreading and Instruction Issue

Multithreading can be seen as another way to increase the "window size," in effect by creating multiple windows – growing the scheduling window "horizontally" rather than "vertically."

The number of threads then becomes one more design parameter in the ILP design space, orthogonal to the method of scheduling within a window. A multithreaded processor could select instructions within a thread for parallel execution using any of the techniques discussed in the previous section, ranging from basic single-instruction issuing to advanced speculation techniques. For instance, a multiscalar machine [25] uses data speculation, although it relies on compiler analysis to arrange threads in such a way as to reduce the probability of mis-prediction. Simultaneous multithreading [31] uses a more modest multi-issue approach, but demonstrated high IPC rates on independent threads.

One can take advantage of the horizontal window growth of a multithreaded processor to reduce the hardware complexity within each window, e.g., by reducing the size of the reorder buffer or by using less-aggressive ILP techniques. For instance, a sequential (single-threaded) superscalar that can look at a window of N instructions can be compared to a multithreaded machine with P threads, where each execution unit on the latter tries to schedule from a window of N/P instructions. The latter may get better performance, because IPC is normally less than linear with respect to the window size; a large number of small windows will likely find more total parallel instructions than a single large window with the same total size.

However, if arbitrary interactions between threads are allowed, then hardware must still check all interactions between all instructions in the windows, leading to hardware complexity comparable to sequential superscalars. The various proposals for multithreaded machines, each with a distinct program execution model, have different restrictions on how threads interact, which in turn affects hardware complexity. We want to design a multithreading model with an eye toward minimizing hardware complexity.

In the discussion which follows, we will concentrate on those aspects of the program execution model which affect inter-thread interactions.

2.2.2 Design Space for Multithreading

One of the most relevant aspects of a multithreaded architecture to parallel scheduling is in the "enabling" of threads – the conditions under which a thread is presumed ready for execution and scheduled for the next available unit. The enabling rule can be *strict* or *non-strict* on data dependence, control dependence, or both. Under a strict enabling rule, a thread cannot become enabled until all corresponding dependences between it and other threads have been satisfied. A non-strict enabling rule allows a thread to begin execution before all data and control information has arrived. This allows threads to start running earlier, but they may stall because of a still-unresolved data dependence, and hardware has the additional burden of checking for these dependences dynamically.

Multithreading models also differ in whether threads are *preemptive* or *non-preemptive*. A preemptive thread can switch from an active state to a "dormant" state and give up the execution resources it is using while waiting to resume execution. A non-preemptive thread, once it has begun executing, will hold onto its resources until execution has terminated, preventing other threads from using the same hardware. There is a tradeoff between these two types of threads. Preemptive threads are more flexible and easier to generate, because one doesn't need to determine which instructions are long-latency operations. However, switching contexts in the middle of thread execution can have a high cost, particularly if OOO execution is involved.

3 The Superstrand Program Execution Model

This section presents the superstrand program execution model. A superstrand program consists of a two-level hierarchy of *threads* and *strands*. Each thread is a collection of strands, which are scheduled atomically, run non-preemptively, and are restricted in the operations they can do in order to reduce hardware complexity. We first present and justify the thread/strand hierarchy, then describe its execution model, and discuss how threads and strands are created, scheduled, activated and deleted.

3.1 Hierarchy of Threads and Strands

In the superstrand execution model, a program is written as a set of *threaded procedures*. Each instantiation of a threaded procedure, called a *thread*, is associated with a separate context of function parameters and local variables. Because the contexts are kept separate, threads can run in parallel – even threads which are instances of the same procedure. Threads can be created and destroyed at run-time, and threads can create new threads.

If threads were written as sequential code, the processor could still exploit parallelism by running several threads at the same time. However, it is not always easy to find "course-grain" parallelism in the form of parallel function calls. The processor could still try to find whatever ILP it could in each thread, using the techniques discussed in section 2.1, but this would lead to the same hardware complexity.

Instead, a superstrand processor uses a smaller unit of scheduling by partitioning a threaded function into smaller units called *strands*. Intuitively, a strand is a group of instructions which, once enabled and activated, should be executed to completion without any stalls due to control dependences or long-latency operations. Strands have a strict enabling rule (see section 2.2), which guarantees that a strand will not start executing until all data used by that strand have been produced (except for intermediate results produced *within* the strand itself). How this rule is enforced is the topic of the next section.

The benefit of the strict enabling rule, from the perspective of the window scheduling discussion in sections 2.1 and 2.2, is that it is no longer necessary for the hardware to check dynamically for dependences between instructions in two different strands. Once the dependences have been satisfied, and the processor begins executing a strand, it only needs to check for dependences among instructions *within* that strand – perhaps using something as simple as scoreboarding if only in-order multi-issue is used to exploit parallelism. This reduces hardware complexity enormously.

Not any random group of instructions is suitable for a strand, even if it obeys a strict enabling rule. Additional restrictions must be placed on what kinds of instructions can go together in one strand:

- Conditional branches either create stalls or require the help of expensive branch prediction hardware. Therefore, only instructions belong to the same "control level" can be grouped into one strand. That is, we mandate that all instructions within a strand have no control dependences with one another, i.e., there are no conditional branch instruction within a strand. As a result, no branch stalls will happen during the execution of a strand. Control flow is handled *outside* of the strand level. A special conditional operation, described in section 4.2, allows strands to selectively enable the execution of other strands according to the value of a conditional expression.

  The notion of "control level" above is closely related to the notion of *control region* in program dependence graphs (PDG) [7]. However, they are not identical. We will illustrate the concept of "control level" in an example to be introduced in Section 4.2. In Section 4, we show how a compiler can naturally follow the "control levels" in the process of strand partitioning.

- A load instruction and any instruction which depends on the load will not be grouped into the same strand, unless it is known with high probability that the load delay can be handled without incurring stalls (either because the compiler knows the object will most likely be in the cache or because enough instructions can be inserted between the load and first use to cover memory latencies). That is, no load stalls should happen during a strand execution.

- All instructions in a strand must belong to the same function.

From the point of view of program execution, the difference between a thread and a strand is that the former is associated with a *frame*, which is a block of memory, plus a collection of registers for holding intermediate data. This frame is used for the storage of local variables, similar to a stack frame. However, since many different threads, each with its own context, can be active simultaneously, these frames cannot be arranged in a conventional stack-like structure. As in some other multithreaded models [11, 21], frames are dynamically allocated from a heap, and are typically linked to one another in a non-linear structure (such as a tree). Frames are linked by passing frame pointers from one thread to another.

All strands within a thread share the frame associated with that thread, and use the frame's local variables or registers for passing data among themselves. Therefore, the activation of a strand does not require the allocation of separate storage. Instead, strands can be viewed as

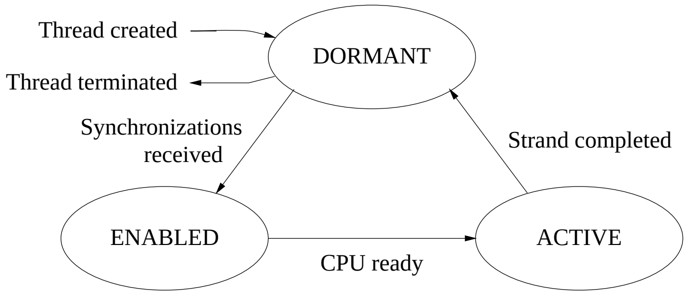


Figure 2: Strand states

frameless threads. A strand is allocated a private *register pad* – a small number of registers that retain intermediate values generated and passed among instructions within a strand. We use the term *register pointer* (rp) to denote the base address of this register pad.

Thus, a strand's total context consists of the private register pad, and several key pointers: components:

- A *frame pointer* (fp), which points to the base of the frame allocated to the thread containing the strand;
- A *thread pointer* (tp), which points to the base of the register set allocated to the thread, and shared among all strands in this thread;
- A *register pointer* (rp), which points to the base of the private register pad;
- An *instruction pointer* (ip), the address of the next instruction to be executed.

The register pad is small and does not need to be initialized, so the only parts of the state that needs to be live when a strand is dormant are the fp, tp and ip. This makes the cost of context switching very low.

3.2 Strand Activation and Scheduling

A necessary condition for a strand to execute is that all data required by the thread have been produced, and all control dependences are satisfied. To enforce this rule, strands must go through a series of states, shown in figure 2. There are three states:

- *DORMANT* strands have not received all data required.
- *ENABLED* strands have received all data required, and are waiting to begin execution, once there is room in the CPU for this strand.
- *ACTIVE* strands have been loaded in the CPU, and are currently executing.

Dormant strands are waiting for data. As soon as the required data is available, the strand is "enabled" and is placed in a *ready queue* to await execution. If the CPU has resources to execute this strand, it removes the strand from the ready queue, allocates a register pad and sets the rp, and begins execution of the strand. The strand is now in the active state. Since the strand is non-preemptive, it stays in the CPU until it finishes, at which point it becomes dormant again and waits for new data.

The CPU can hold a limited number of strands in the active state, depending on its hardware resources. When a strand terminates and is removed by the CPU (changed to the dormant state), the CPU goes to the ready queue to dequeue and activate the next strand. A *strand scheduling policy* determines the order in which threads are activated. A simple FIFO scheduler would activate strands in the order in which they become enabled. Other scheduling policies can be used to try to optimize strand ordering for locality, but a full discussion of this issue is beyond the scope of this paper.

When a threaded function is called, a thread and associated frame are created, parameters to the function are placed in the frame, and an initial set of strands is created. All strands begin in the dormant state, except for one strand called the *initialization strand*. This strand is automatically enabled because it is assumed that it will only use the function parameters, which are already available. The initialization strands will pass data to other strands, enabling some of them in the process.

The mechanism for determining when a strand has received all its data is based on the following principle. Each strand is associated with a *synchronization count* (or *sync count*), which counts the number of dependences on which this strand depends. If another strand produces a datum or control condition on which this strand depends, the sync count for this strand is decremented. When the sync count reaches 0, all dependences have been resolved, and the strand can change to the enabled state. When an active strand becomes dormant, the sync count is reset to the original count, so that it can begin counting synchronization events again. This simple mechanism has its origin on the static dataflow model of computation [5], but further extended in the EARTH multithreaded execution model [11].

4 Program Representation and Strand Partition

In this section, we will outline a program presentation, called *program dependence flow graph* (PDFG), and describe the strand partitioning strategies for PDFG graphs. In Section 4.1, we illustrate the PDFG representation and its operation semantics. In Section 4.2, we show how to partition a PDFG graph into strands, and demonstrate the overall strand execution model through the partitioning results.

4.1 Program Dependence Flow Graph

Program dependence flow graph (PDFG) is an executable program representation, in which the control structure and data dependence of a program are represented in a hierarchical fashion.

A PDFG graph can be built easily based on the *program structure tree* (PST) representation. A PST tree of a program is a hierarchical representation of the control structure of the program [15]. For a structural program, the PST-tree representation is the same as the AST representation. For example, the SIMPLE AST is a PST representation used in our compiler for partitioning structural programs into threads [10, 26]. A PDFG graph is a PST tree, modified and augmented with additional nodes and edges to represent precedence constraints caused by data and control dependences. Moreover, with the newly added constraints a PDFG graph has well defined operational semantics and can be executed directly.

Let us illustrate our program representation with an example. Figure 3 is a program that computes the weight of a symbol. The weight is proportional to the number of times a symbol appears in two linked lists. During each `for-loop` iteration, a symbol is retrieved from a symbol array (`Buffer[]`), and then this symbol is checked against two linked lists, denoted by `list_header` and `list_header1`. Whenever a symbol is detected in a list, its partial weight is incremented by its original weight (`w`). After the traversal of two linked lists is finished, the sum of two partial weights (`r and r1`) is computed and assigned back to the weight array (`Weight`).

The PST tree of the example program is shown in Figure 4, in which nodes represent *single-entry-single-exit* (SESE) regions of the program and edges represent the nesting of regions. A simple statement (such as S_1 and S_2) forms a *trivial SESE region*, and is represented as a leaf node. A compound statement (such as S_0, S_5, and S_{12}) contains one or more nested regions, is represented by an internal node. Such an internal node is also called a region node. Similar to the concept of *basic blocks*, a *basic region* is a sequence of regions (also called *sequentially composed regions* in [15]) such that the nodes in the same basic region have the same parent in the PST with the identical control dependence. In the figure, dash-edged rectangles outline these basic regions. For example, the compound *while-loop* statement S_5 contains one subcomponent (loop body) which consists of three nodes: S_6, S_7, and S_9. These three nodes form a basic region. Furthermore, they share the same parent node S_5, which all three are control dependent on.

Now we see how to modify the program structure tree to construct a *program dependence flow graph (PDFG).*

First, we introduce data dependence edges between nodes within the same basic region. In other words, we build a data dependence graph (DDG) for each basic region. In Figure 5, two DDGs are shown where DDG edges are drawn in solid lines. One DDG is for the basic region corresponding to the `for-loop` body (at statement S_0), and the other for the basic region corresponding to the `while-loop` body (at statement S_5). Moreover, to maintain the SESE feature in the following step of refining the control structure of PDFG graphs, a region

```
s0:  for (i=0; i<BuferSize; i++) {
s1:    symbol = Buffer[i];
s2:    w      = Weight[i];
s3:    r      = 0;
s4:    list   = list_header;
s5:    while (list != NULL) {
s6:      entry = list -> symbol;
s7:      if (entry == symbol) {
s8:         r = r + w;
         }
s9:     list = list -> next;
       }
s10:   r1     = 0;
s11:   list1  = list_header1;
s12:   while (list1 != NULL) {
s13:     entry1 = list1 -> symbol;
s14:     if (entry1 == symbol) {
s15:         r1 = r1 + w;
         }
s16:     list1 = list1 -> next;
       }
s17:   Weight[i] = r + r1 ;
     }
```

Figure 3: A fragment of C code

node (corresponding to a compound statement) is split into a pair of nodes: a "begin" and a "end" node as illustrated in Figure 5. Due to space limitation, DDGs for other basic regions are omitted from the figure.

Second, we replace PST-containing edges with control flow (transfer) edges (CFG edges), and use control operators (*merge and switch*) to connect two nested regions so that the resulting program representation becomes executable. More specifically, CFG edges and control operators (*switch* or *merge*) are used to connect the region-begin and region-end nodes with their closely nested region(s) as illustrated in Figure 5, where control edges are drawn in dashed lines. For example S_{7_1} is a switch node, which represents the conditional part of `if` statement (i.e. statement S_7 in the original code). It has one input control edge from the region-begin node of S_7, and two output edges. The first is connected to S_8, which is the first node of its closely nested region corresponding to the "then arm" of the conditional. The second output is connected directly to the merge node X (since the "else arm" is empty). The output of X goes back to the region-end of S_7. Similarly, for the `for-loop` region denoted by S_0, the switch node S_{0_1} (loop condition) has also two output edges: one leads to S_1, which is the first node of the closely nested region (loop-body); the other goes back to the region-end of S_0. In addition, a merger node is inserted to combine control edges from the region-begin node of S_0 and the back edge of the loop iteration (from S_{0_2}). Note that the above procedure of adding CFG edges can easily be formalized so that it can be applied to all nested regions.

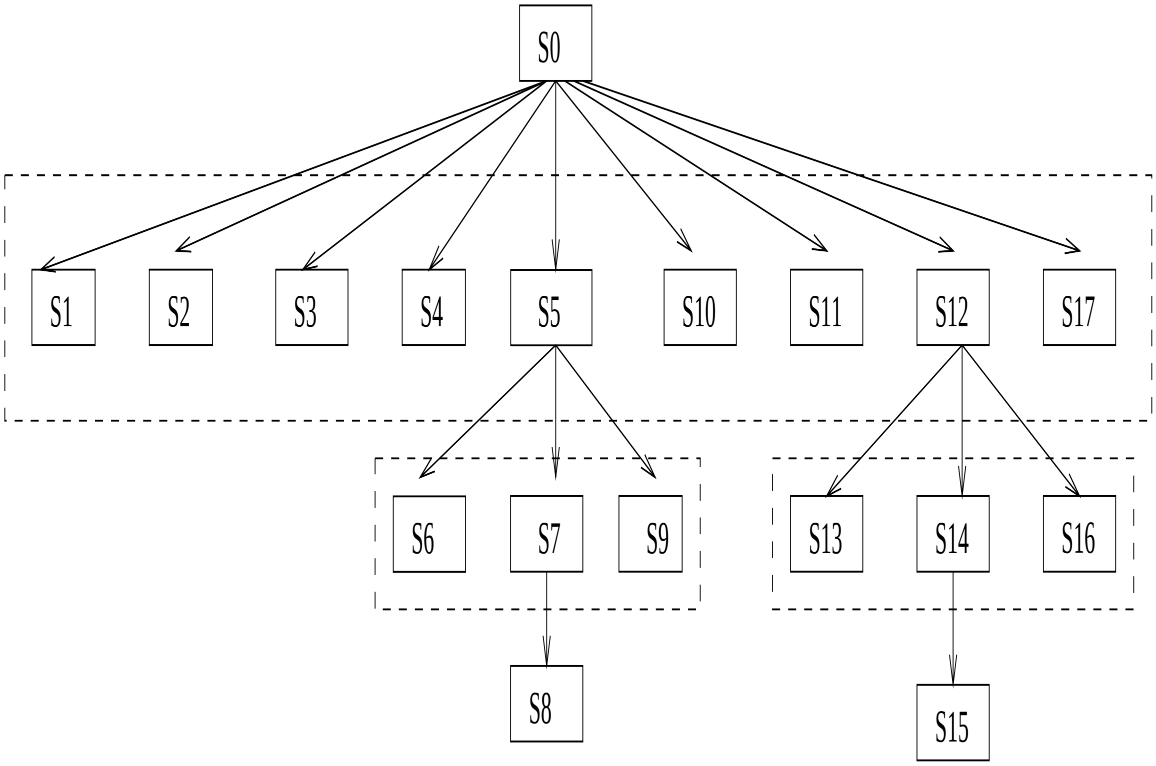


Figure 4: The PST tree of the example C code

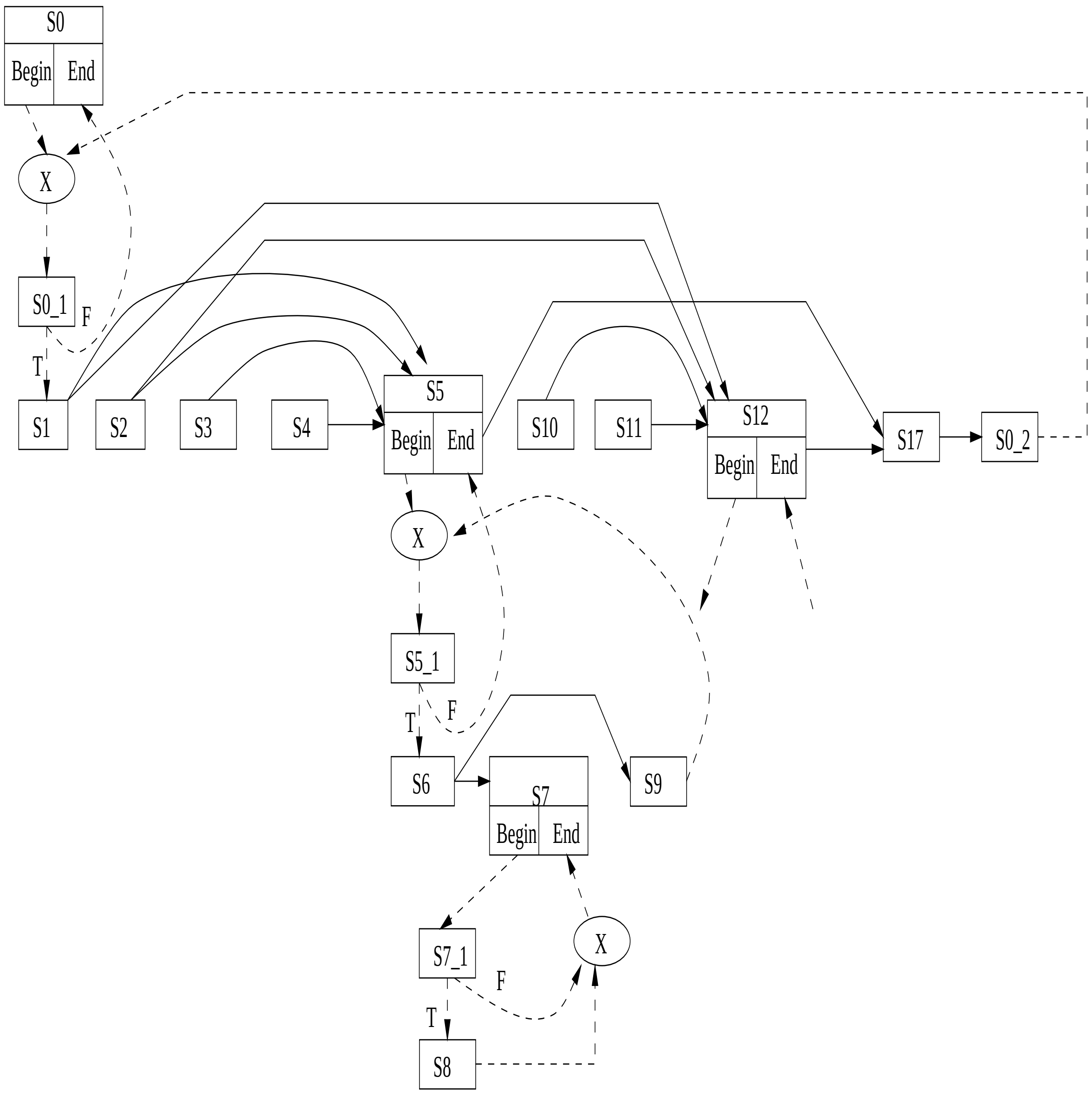
However, although S_{17} does not have any data dependence with S_{0_1} in the same loop iteration, we still introduce an artificial data dependence edges to ensure that the next iteration will not start until all operations in the current iteration complete. Such sequential constraints can be relaxed if the overlapped execution of different loop iterations is to be exploited. In this paper, however, we assume that loop unrolling will be performed to expose the parallelism in order to simplify our discussion.

In summary, a PDFG graph consists of two parts: a set of DDGs built for each basic region, and a CFG graph connecting nested regions. Intuitively, there is no DDG edges crossing regions, and control flow enters a SESE region at the beginning and leaves at the end.

As we showed earlier, a PDFG graph of program P can be built from its corresponding PST tree. Thus, the generality of PDFG graph follows the same property of PST tree.

The operational semantics of PDFG nodes are specified by the following “firing rules”:

- All normal nodes (e.g. not a switch or merge node) become enabled when all input signals have arrived. The firing of an enabled node will place signals at each of its output arcs (both data and control flow arcs).



DDG edges connect nodes in the same SESE region and CFG edges flow between nested regions. Node X denotes a nondeterminate merge which allows only one control transfer passing through.

Figure 5: Program Flow Dependence Graph

- A switch node becomes enabled when all inputs signals have arrived. The firing of an enabled switch node will place a signal on one of its two output arcs, depending on the result of the conditional expression.

- A merge node (X) becomes enabled if anyone of its input signal arrives. The firing of an enabled merge node will simply send a signal to its output arc.

The firing rules also apply to the region *begin* and *end* nodes. In firing, a *begin* node sends control signals to its closely nested regions while an *end* node sends data signals to the nodes that are data dependent on it within the same basic region.

4.2 Strand Partitioning

In this section, we briefly outline how the nodes in the PDFG can be grouped into strands. One of key problems in strand partition is to decide which types of edges should be cut so that source and destination nodes are put into different strands. In this paper, we define two types of edges as the candidate to be cut:

1. Remote DDG edges: DDG edges emitting from the nodes which have long latency operations. These nodes are called *remote* nodes and will be executed in a *split-phase* manner. For example, nodes which access *array* and *heap* may be *remote* because of memory latencies. In Figure 5 node S_1 and S_2 are remote since they contain either array or heap access.
2. CFG edges connecting different nested regions. For example, node S_{5_1} and S_6 are connected by a control flow edge (true arm). After strand partition, they must be put in two different strands.

Figure 6 gives the result of strand partitioning for the program listed in Figure 5. A strand is delimited by a dashed rectangular and is labeled with its number. Because merge nodes (X) do not perform any "real" operation. they do not have any impact on strand partitioning. They and the edges incident on them are omitted for clearness. In Figure 6, all remote nodes are showed in shadow and they are separated from the nodes which rely on them. For example, S_1 and S_2 are in strand 2 while S_5 and S_{12}, which rely on S_1 and S_2, are in strand 3.

The hierarchical representation facilitates greatly for the strand partitioning process since remote DDG edges and CFG edges are always in different hierarchical levels. Thus, we can first cut DDG edges within each basic region, and then cut CFG edges between two nested regions. The description of strand partition algorithms is beyond the scope of this paper. We simply list the partitioning result in Figure 6.

In Figure 6, we use circled numbers to represent how many remote DDG edges exist between two strands since only nodes connected by remote edge need synchronization. For example, there are two remote DDG edges emitting from strand 2 to strand 3. They represent the union of original DDG edges of (S_1, S_5),(S_2, S_5), and (S_1, S_{12}), (S_2, S_{12}).

In Figure 6, some optimizations are applied to improve the partitioning result. For example, S_7 and S_{7_1} are merged into one strand since there is an single edge connecting them, and this

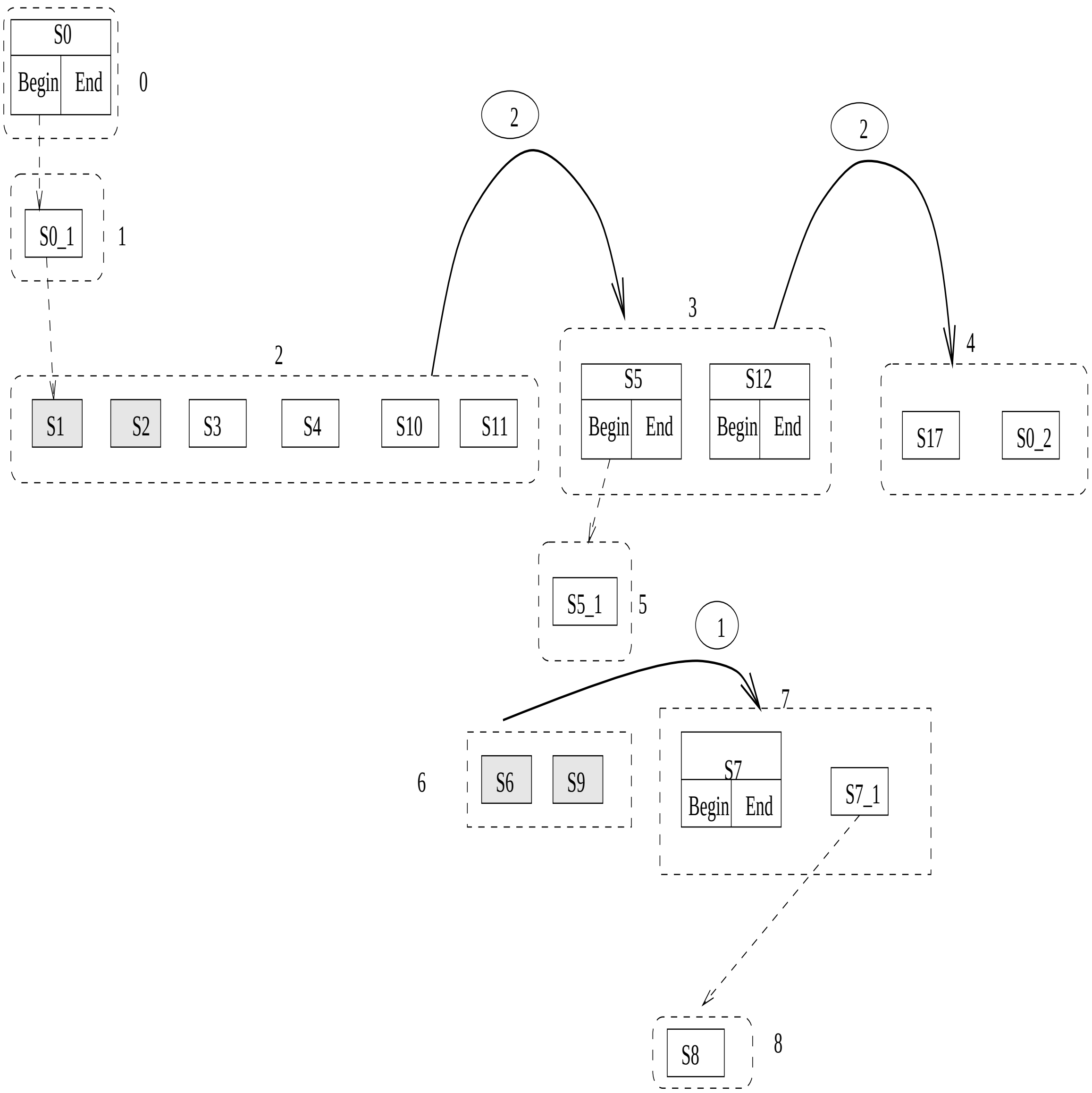


Figure 6: Partitioned Strands

edge does nothing but starting node S_{7_1} unconditionally. However, putting S_6 and S_9 together relies on the fact that the false anti-dependence can be clearly disambiguated at compiler time.

Figure 7 lists the corresponding *stranded* code and Table 1 lists synchronization signals among strands. Please notice that merge and region-end nodes are eliminated since their existence is to help compute synchronization signals among threads.

A split-phase operation $CondSync(P, i, j)$ is introduced in the final code. This primitive sends a signal to $STRAND_i$ if condition P is true; otherwise to $STRAND_j$. Some partitioning

```
s0:   i=0; SYNC(ST_1);
      END_STRAND();
STRAND_1:
s0_1: CondSync((i<BuferSize),2, 9)
      END_STRAND();
STRAND_2:
s1:   symbol=Buffer[addr];
s2:   w      = Weight[i];
s3:   r      = 0;
s4:   list   = list_header;
s10:  r1     = 0;
s11:  list1  = list_header1;
      END_STRAND();
STRAND_3:
s5:   SYNC(ST_5);
s12:  SYNC(ST_S12); /* Not Shown */
      END_STRAND();
STRAND_5:
s5_1: CondSync((list!=NULL),6, 4);
      END_STRAND();
STRAND_6:
s6:   entry=list->symbol;
s9:   list = list->next;
      END_STRAND();
STRAND_7:
s7:   CondSync((entry==symbol),8,5);
      END_STRAND();
STRAND_8:
s8:   r = r + w ;
      END_STRAND();
STRAND_4:
s_17  Weight[i] = r + r1 ;
s0_2: i = i + 1;
      END_STRAND();

STRAND_9:
```

Figure 7: Threaded C code with explicit synchronization

heuristics can be used to leave *CondSync* at the end of a strand so that the switching of strand must happen after this operation is issued.

The first two columns of Table 1 is actually coded in the stranded executable code, and they are used to initialize the strand synchronization. The firing of a strand is managed by hardware. A strand starts its execution after all its needed data or control signals are ready. For example, strand 2 will start its execution after it receives a signal from strand 1 (true-arm). Similarly strand 3 can start its execution only after it receives two data signals from strand 2. Even though the two types of signals actually coexist, representing two different dependences:

Strand	SYNC Count	From	Comments
1	0	{ }	First strand
2	1	$\{ST_1\}$	
3	2	$\{ST_2\}$	
4	2	$\{ST_3,\}$ $\{S_{12}\}$	Wait for two Loops to finish and start next iteration
5	1	$\{ST_3\}$	While loop header
6	1	$\{ST_5\}$	
7	2	$\{ST_6\}$	CondSync
8	1	$\{ST_7\}$	Next while loop
9	1	$\{S_1\}$	Exit Loop

Table 1: Synchronization Table

control and data dependences, underlying hardware will treat them in the same way as the dataflow dependence. Thus, dataflow firing rules introduced earlier can be applied uniformly to govern the strand synchronization.

5 A Superstrand Architecture Model

So far we have presented a model for the representation of a program (even a conventional sequential program) as a hierarchy of threads, strands and individual instructions, called a PDFG. We have also shown how a PDFG can be generated from a program and partitioned into strands by a compiler. We now consider what kinds of processor architectures can support efficient execution of PDFG programs.

This section is divided into two parts. The first covers, in broad terms, the architecture of a superstrand processor, and discusses ways in which parallelism can be exploited by such a processor. The second part describes the mechanisms used to support the scheduling of threads and strands under our execution model. Together, these mechanisms comprise the *Rigged Scoreboarding* (RS).

5.1 Architecture Overview

As described in previous sections, we view a program as being divided into a hierarchy of threads and strands. In this section, we focus on the architecture support for strands as they consist of the main architecture innovations of the superstrand architecture. A strand is ready for execution when all control and data dependence it requires has been satisfied, and it begins execution when resources have been allocated for it. Strands signal other strands and threads using *sync* operations to signal that operands are ready.

Instructions within a strand are executed using ordinary sequential semantics. Thus, the execution pipeline of a superstrand processor (e.g., ALU, LOAD/STORE, etc.) would look similar to a conventional processor. Instructions would pass through fetch, decode, operand access, execute and writeback stages, with load/store buffers used where needed. A conventional register scoreboard would be used to detect data dependences between instructions in the same strand.

There are several key differences between a superstrand processor and a conventional processor, mainly in the scheduling of instructions. First, there are no conventional branch or jump instructions. Control flow is handled according to the superstrand scheduling model. Strands execute *SYNC* instructions to synchronize other strands, and a strand which has received all necessary synchronizations can be placed in a *ready queue* of enabled strands. Loads with an expected long latency are implemented with split phase non blocking *LDSYNC* instructions. Conditionals are implemented with conditional *CONDSYNC* instructions (as explained in section 4.2), where a strand can decide whether or not to synchronize another strand based on the state of a condition code. (Small conditional clauses – 1 or 2 instructions – are probably best handled by converting control dependences into data dependences, i.e., predicated instructions.)

This mechanism allows a smooth pipelining without branch delays. The instruction fetch unit can begin fetching instructions from the lead strand in the ready queue even while a previous strand is still executing, so that when the previous strand terminates, the new strand can be merged into the stream. There will still be a delay from the time a predicate is evaluated to the time a control-dependent clause begins execution, but this time can be put to good use, either because the strand which evaluated the predicate still has other work to do, or the processor can fetch another strand from the ready queue.

Thus, the superstrand processor can eliminate pipeline bubbles created by mispredicted branches, and can get rid of expensive branch-prediction hardware. It does not eliminate the latency imposed by a conditional branch, but merely *hides* it with other strands.

A second key difference between superstrand and conventional processors is the way parallelism is exploited. Superscalar processors adhere to sequential code semantics, and attempt to schedule operations in parallel, dynamically using expensive hardware techniques such as out-of-order execution and register renaming. Since a superstrand processor has program parallelism revealed to it by the compiler, it can dispense with these mechanisms and exploit the parallelism directly.

One way to exploit this parallelism would be to have multiple strands executing at the same time. There would be a set of program counters, each corresponding to one executing strand. The processor would try to issue instructions from as many strands as possible. If the compiler adheres to the superstrand execution model, then the strands are guaranteed not to interfere with each other. Therefore, no hardware is needed to detect hazards between strands; only a simple register scoreboarding scheme is needed for dependences within strands.

The processor may execute a single instruction from each strand, or it may try to issue several consecutive instructions from one strand in a cycle, relying on the local scoreboarding to detect intra-strand hazards. Of course, the number of instructions can be issued from one strand is limited without the more expensive mechanisms, such as support for out-of-order execution. Nevertheless, a simple superscalar issuing scheme coupled with multiple strands has been shown to improve total issue rate significantly [31].

5.2 Strand Synchronization and Scheduling Using Explicit Dependence Information

The key element separating the superstrand architecture from superscalar and other multithreading architectures is the maintenance and use of "explicit dependence" information in supporting the synchronization and scheduling of strands according to the superstrand execution model, as described in section 4. To this end, we have introduced *rigged scoreboarding (RS)*, and present its design in this section. We call this mechanism rigged scoreboarding because we use explicit dependence information discovered by the compiler whereas register scoreboarding uses implicit dependence information in the sequential code discovered and resolved at runtime.

Before we present our solution, we discuss limitations or shortcomings of the synchronization mechanisms of dataflow machines and dataflow-derivative multithreaded machines (including our work in the past [9, 11]).

Perhaps a natural question is: Why not use conventional register scoreboarding to implement inter-strand synchronization ?

Scoreboarding — at least in the general sense – is a "dynamic" hazard resolution mechanism. Dependence resolution implies two aspects: (1) hazard detection, (2) hazard handling. General register scoreboarding will do both the "detection" and "handling" dynamically at runtime. In our strand model, we assume that hazard (dependence) analysis and detection are mainly done at compile time . Our inter-strand synchronization is based on the fact that a compiler has explicit knowledge of dependence, and can place the synchronization at the strategic "cut" point of the program graph! Therefore, by maintaining explicit dependence information between strands, our solution should be simpler – since we should not need the full power of "detection" anymore. Furthermore, some of these inter-strand synchronizations may not even be register based, thus cannot be handled by register scoreboarding.

Under the superstrand synchronization model, a strand is not ready for execution until all operands have been produced. In dataflow machines, and in dataflow-like multithreaded machines (such as EARTH), this requirement is handled by associating a counter with each instruction or strand. The counter is initialized with the number of synchronizations needed by that strand, and each *sync* instruction directed toward that strand causes the counter to be decremented, until it reaches zero, which causes the strand to be put in the queue for execution.

If a program may be partitioned into a large number of strands, then a large number of counters must be allocated, which leads to two problems. First, it may be difficult to keep so many counters close to the processor state. Counters, like registers and other high-speed devices, are a precious resource to be allocated carefully. Second, a large strand address space implies there are many bits in a strand field.

We note, however, that in most cases, if a thread has many strands, they will not all be running in parallel, but rather will have dependences between them. This means that only a subset of strands will be active at a given point in time. It follows that only a subset of strands can receive synchronization events at that time, and therefore, only those counters associated with those strands need to be in the fast processor state.

This suggests an efficient implementation of the RS module. For each executing thread, there is a set of synchronization counters. In many cases, the number of available counters is smaller than the total number of strands, so they must be reused, in the same way that ordinary registers are reused. The compiler can do this with a graph coloring algorithm, analogous to the way interference graphs are used to perform register allocation.

An alternate way to implement strand synchronization is to treat each synchronization event as a separate entity, and to map these events to strand synchronizations through an associative mapping which takes the firing rules of each strand into account.

The general scheme is shown in figure 8.

There is a row of bits called *synchronization register bits* (SRB). Beneath the SRB row is a matrix with one column for each SRB. Each row in this matrix corresponds to a strand which is eligible to receive synchronizations, and the entries in that row represent the firing rules for that strand (i.e., which synchronizations are required for this strand). Each row is associated with the ID of one strand (SID). In RS figures in this section, a symbol $\times$ in row i, column j means that SRB_j must be synchronized before the strand corresponding to row i can be enabled. This is conjunctive; if row i has multiple bits set, all corresponding SRB bits must be synchronized before the strand is enabled.

Given the quadratic hardware complexity, both the number of columns (SRBs) and the number of rows should be small. The number of SRBs can be kept small by using one SRB for more than one strand. A synchronization event has a limited lifetime; it is only live from the time the *sync* instruction is executed to the time where the strand is enabled and placed in the ready queue. Therefore, as with data registers, SRBs may be reused, and the compiler may construct a mapping from individual synchronization events to SRBs using a graph coloring algorithm based on interference graphs.

If there are more strands than rows, then only a subset of the strands can be in the active matrix at one time. Since synchronizations can come from any strand executing, every strand

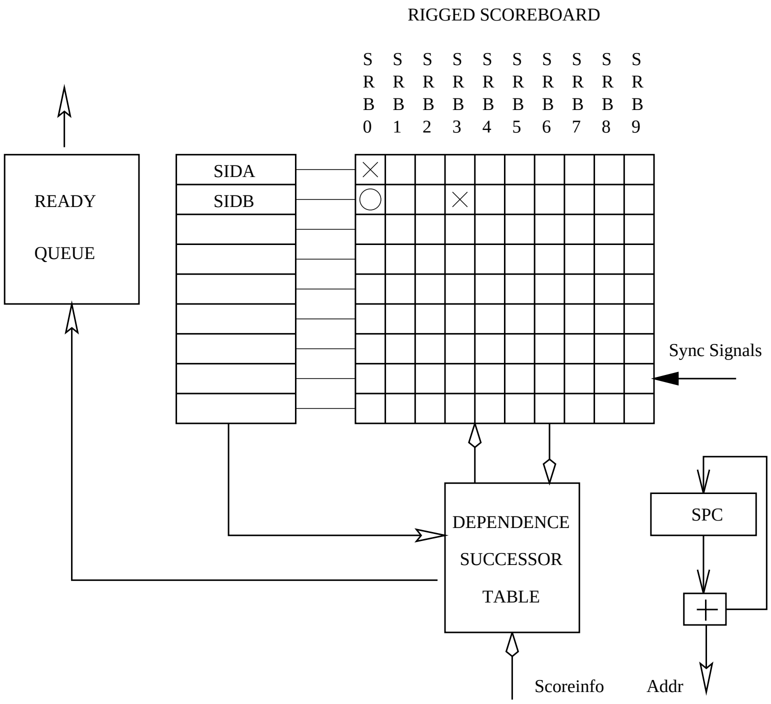


Figure 8: Rigged Scoreboarding (RS)

which can receive a synchronization directly from a currently executing strand must be in the matrix. When a strand begins execution, all strands which may receive synchronizations from that strand must be placed in the matrix, so that when the thread executes a *sync* instruction, there will be something for the *sync* operation to activate.

All strands in the thread are kept in a *dependence successor table* (bottom of figure 8). This table is indexed by the strand ID (SID), and each entry contains the PC address of the first instruction of the strand, the "firing rules" corresponding to a row in the matrix, and a list of the SIDs of all possible strands which may be synchronized by this strand. When the RS fires a strand, it must look up the strand in the dependence successor table to get the list of dependence successor strands, and place each of these strands in the active matrix.

Since RS will not allow a strand to begin execution until this operation is complete, there can be delays from the time RS detects a row match in the matrix and the time the entry in the dependence successor table is fetched. Therefore, it isn't essential that all strands be quickly accessible in the dependence successor table. Indeed, the dependence successor table could be a data structure in regular memory, with a small cache of dependence successor table entries kept near the RS matrix for speed, or the RS could simply keep the dependence successor entries in instruction memory and read them through the instruction cache.

Since the firing of a strand causes new entries to be brought into the matrix, old entries must be removed or else the matrix will overflow. When a strand is enabled, it is removed from the matrix, since it has received all required synchronizations and is not expecting to receive any more. A second, more complicated rule concerns strands which are conditionally synchronized. For example, figure 14 shows strand 5, which either syncs strand 6 or strand 4, depending on the condition code. At the time strand 5 begins execution, strand 6 and strand 4 must both be loaded into the matrix, since either may be synchronized. However, if one of them receives a *sync*, it is clear that the other one will *not* be synchronized in the current iteration, and can be deleted.

To handle this special case, there is another type of entry in the matrix row, shown in our diagrams as the symbol $\bigcirc$. Such an entry in column j means that if SRB_j is synchronized, the corresponding strand will *not* be enabled and can be deleted.

Finally, SRB bits themselves must be cleared. Usually, if a strand is enabled, the SRB bits which caused the enabling can be cleared, since they have served their function of synchronizing the strand. But this is not always the case. It is possible for one SRB to synchronize more than one strand; this would correspond to a column with more than one $\times$ and/or $\bigcirc$ entry. This would represent a strand synchronizing more than one dependent strand, i.e., a strand with a "fan-out" of two or more. (It is this ability to share a synchronization bit among multiple strands that makes our scoreboarding technique more attractive than the counter scheme described earlier.) Therefore, bit SRB_j can't be cleared until all rows with an $\times$ in column j have been removed.

In summary, the RS operates according to the following rules:

1. When a strand executes a *sync* instruction specifying SRB_j, set that SRB bit.
2. Associatively search all rows in the matrix for matches with the current state of the SRBs. For each row i in the matrix:
   - If row i, column j is $\bigcirc$, remove row i from the matrix without any enabling; reset all $\times$ and $\bigcirc$ entries.
   - If every column j (in row i) which is $\times$ is matched by a set bit in SRB_j, fetch the SID for row i, and use this to look up the dependence successor entry in the dependence successor table. Once the SID has been fetched, remove row i, and reset all $\times$ and $\bigcirc$ entries.
3. For each SRB_k which contributed to an enabling or removal of a row i, if there are no $\times$ symbols in column k left, clear SRB_k.
4. If the SID for row i was fetched, then perform the following additional steps:
   (a) Once the dependence successor table entry for the given SID has been fetched, examine the dependence successor list of that entry, and fetch each dependence successor SID in the list. For each dependence successor fetched, assign a row in the table, setting the SID and firing rule entries according to the dependence successor entry.

(b) Once all dpendence successors have been fetched and loaded into the matrix, fetch the PC address for the given SID and place it in the ready queue.

Note that, with enough entries in the matrix, the RS will never deadlock. We leave it to the readers to verify this as an exercise.

In Appendix A, we show how RS works using the program example as a case study.

6 Related Work

Recently, a number of other architectures have been proposed to deal with the increasing demand for higher ILP by increasing on chip parallelism.

Tera [2] and Alewife [1] are classical multithreading architectures which intend to hide resource latencies as incurred by functional units and memory. To achieve this goal, fast context switching is required between threads.

A natural extension to conventional multithreading architectures is Simultaneous Multithreading [31]. The authors propose an architecture where several independent threads are allowed to issue instructions to a superscalar execution engine in a single cycle. This technique attacks horizontal waste in addition to vertical waste tackled by previous architectures.

Franklin and Sohi [8] address the complexity issue of the centralized instruction window by splitting the window into basic windows. The Multiscalar architecture [25] follows this approach and intends to improve superscalar performance by scheduling "task" units with the provision of aggressive control, memory and data speculation.

In [19] the limitations of superscalar architectures are tackled by proposing single-chip Multiprocessors. Their decision to opt for multiprocessing on chip is driven by complexity concerns of other architectural approaches.

Vajapeyam and Mitra [32] improve superscalar instruction dispatch by taken dynamic traces as in Trace Caches [20] to subdivide the instruction window.

Compiling for multithreaded architectures is a relatively new field. Most of work has been done for functional languages such as **Sisal** and **Id**. The most influential algorithms include the *dependence set [12]*, *demand set [23]*, *iterated partitioning [4]*, and *separation constraint partitioning [22]*. We also developed partitioning algorithms for imperative languages in which various heuristics were designed based on a cost model for multithreaded architectures [27, 10].

7 Conclusions and Future Work

In this paper we have proposed a novel scheme to deal with centralized instruction window complexity in the context of a multithreaded architecture. Our scheme partitions the instruction window hierarchically in several levels. In this paper we introduced the intra- and inter-strand level. Each level book keeps dependencies by a simple scoreboarding mechanism determined statically at compile time by an extended register coloring mechanism.

The problem of previous dataflow architectures in providing enough resources to maintain the complete dataflow graph at runtime has been overcome by mapping parts of the dataflow graph time interleaved into RS. To our knowledge, current and proposed multithreading architectures do not have any provision for fast, low overhead communication between threads at the register level while next generation superscalar and VLIW architectures are limited by the parallelism exposed in a contiguous (yet split) instruction window.

In the future, we intend to validate the above proposed architecture with extensive and accurate simulation on our in-house SEMi simulator [28]. Further on, we intend to extend our model with additional window hierarchies. Alternative RS designs - the "counter" approach e.g as shortly exposed in section 5 - will be investigated. Memory accesses within this architecture will be researched. A dynamic dataflow approach in this context might be considered.

Finally, we are interested how far we can push the ILP envelope without speculation - a key factor for increased complexity in modern processor design.

Acknowledgments

This work has been supported in part by DARPA, NSA, NSF (NSF Grant No. is ASC-9612105), and NASA. We would like to acknowledge useful discussions on program execution models with Dr. Tom Sterling and other HTMT team members. Discussions on RSFQ technology with Prof. Likharev and his team of SUNY at Stony Brook has been very helpful. Finally we thank other members of the HTMT group in the Computer Architecture and Parallel Systems Laboratory, Department of Electrical and Computer Engineering, University of Delaware.

References

[1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife machine: Architecture and performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 23(2), May 1995.

[2] Gail Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. Presented at the Workshop on Multithreaded Computers, held at *Supercomputing '91*, Albuquerque, New Mexico, November 1991.

[3] Todd M. Austin and Gurindar S. Sohi. Dynamic dependency analysis of ordinary programs. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 342–351, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 20(2), May 1992.

[4] David E. Culler, Seth C. Goldstein, Klaus E. Schauser, and Thorsten von Eicken. TAM – a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, July 1993.

[5] Jack B. Dennis. The evolution of "static" data-flow architecture. In Jean-Luc Gaudiot and Lubomir Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 2, pages 35–91. Prentice-Hall, Englewood Cliffs, New Jersey, 1991. Book contains papers presented at the First Workshop on Data-Flow Computing, held in conjunction with the 16th Annual International Symposium on Computer Architecture in Eilat, Israel, May 1989.

[6] Jack B. Dennis and Guang R. Gao. Multithreaded architectures: Principles, projects, and issues. In Iannucci et al. [13], chapter 1, pages 1–72. Book contains papers presented at the Workshop on Multithreaded Computers, held in conjunction with Supercomputing '91 in Albuquerque, New Mexico, November 1991.

[7] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[8] Manoj Franklin and Gurindar S. Sohi. The expandable split window paradigm for exploiting fine-grained parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 58–67, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 20(2), May 1992.

[9] G. R. Gao. An efficient hybrid dataflow architecture model. *Journal of Parallel and Distributed Computing*, 19(4):293–307, December 1993.

[10] Laurie J. Hendren, Xinan Tang, Yingchun Zhu, Guang R. Gao, Xun Xue, Haiying Cai, and Pierre Ouellet. Compiling C for the EARTH multithreaded architecture. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 12–23, Boston, Massachusetts, October 20–23, 1996. IEEE Computer Society Press.

[11] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Guang R. Gao, and Laurie J. Hendren. A study of the EARTH-MANNA multithreaded system. *International Journal of Parallel Programming*, 24(4):319–347, August 1996.

[12] Robert A. Iannucci. A dataflow/von Neumann hybrid architecture. Technical Report MIT/LCS/TR-418, MIT Laboratory for Computer Science, Cambridge, Massachusetts, July 1988. PhD thesis, May 1988.

[13] Robert A. Iannucci, Guang R. Gao, Robert H. Halstead, Jr., and Burton Smith, editors. *Multithreaded Computer Architecture: A Summary of the State of the Art.* Kluwer Academic Publishers, Norwell, Massachusetts, 1994. Book contains papers presented at the Workshop on Multithreaded Computers, held in conjunction with Supercomputing '91 in Albuquerque, New Mexico, November 1991.

[14] MIPS Technologies Inc. *R10000 Microprocessor User's Manual-Version 2.0.* MIPS Technologies Inc., December 1996. Version 2.0.

[15] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 171–185, Orlando, Florida, June 20–24, 1994. *SIGPLAN Notices,* 29(6), June 1994.

[16] Gerry Kane and HP. *PA-RISC 2.0 Architecture book.* HP, February 1996. Version 2.0.

[17] Luis A. Lozano C. and Guang R. Gao. Exploiting short-lived variables in superscalar processors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 292–302, Ann Arbor, Michigan, November 29–December1, 1995. IEEE-CS TC-MICRO and ACM SIGMICRO.

[18] Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 181–193, Denver, Colorado, June 2–4, 1997. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 25(2), May 1997.

[19] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, Cambridge, Massachusetts, October 1–5, 1996. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society. *Computer Architecture News,* 24, October 1996; *Operating Systems Review*, 30(5), December 1996; *SIGPLAN Notices,* 31(9), September 1996.

[20] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 24–34, Paris, France, December 2–4, 1996. IEEE-CS TC-MICRO and ACM SIGMICRO.

[21] James Rumbaugh. A data flow multiprocessor. *IEEE Transactions on Computers*, 26(2):138–146, February 1977.

[22] Klaus E. Schauser, David E. Culler, and Seth C. Goldstein. Separation constraint partitioning — A new algorithm for partitioning non-strict programs into sequential threads. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 259–271, San Francisco, California, January 22–25, 1995.

[23] Klaus Eric Schauser, David E. Culler, and Thorsten von Eiken. Compiler-controlled multithreading for lenient parallel languages. Report No. UCB/CSD 91/640, Computer Science Division, University of California at Berkeley, 1991.

[24] Raúl Silvera, Jian Wang, Guang R. Gao, and R. Govindarajan. A register pressure sensitive instruction scheduler for dynamic issue processors. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 78–89, San Francisco, California, November 10–14, 1997. IEEE Computer Society Press.

[25] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 23(2), May 1995.

[26] Xinan Tang, Rakesh Ghiya, Laurie J. Hendren, and Guang R. Gao. Heap analysis and optimizations for threaded programs. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 14–25, San Francisco, California, November 10–14, 1997. IEEE Computer Society Press.

[27] Xinan Tang, Jian Wang, Kevin B. Theobald, and Guang R. Gao. Thread partitioning and scheduling based on cost model. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 272–281, Newport, Rhode Island, June 22–25, 1997. SIGACT/SIGARCH and EATCS.

[28] Kevin B. Theobald. SEMi: A simulator for EARTH, MANNA, and the i860 (version 0.15). ACAPS Technical Note 43 (Revised), School of Computer Science, McGill University, Montréal, Québec, October 1996.

[29] J. E. Thornton. *Design of a Computer: The Control Data 6600.* Scott, Foresman, and Co., 1970.

[30] Garold S. Tjaden and Michael J. Flynn. Detection and parallel execution of independent instructions. *IEEE Transactions on Computers*, 19(10):889–895, October 1970.

[31] 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*, pages 392–403, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 23(2), May 1995.

[32] Sriram Vajapeyam and Tulika Mitra. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 1–12, Denver, Colorado, June 2–4, 1997. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 25(2), May 1997.

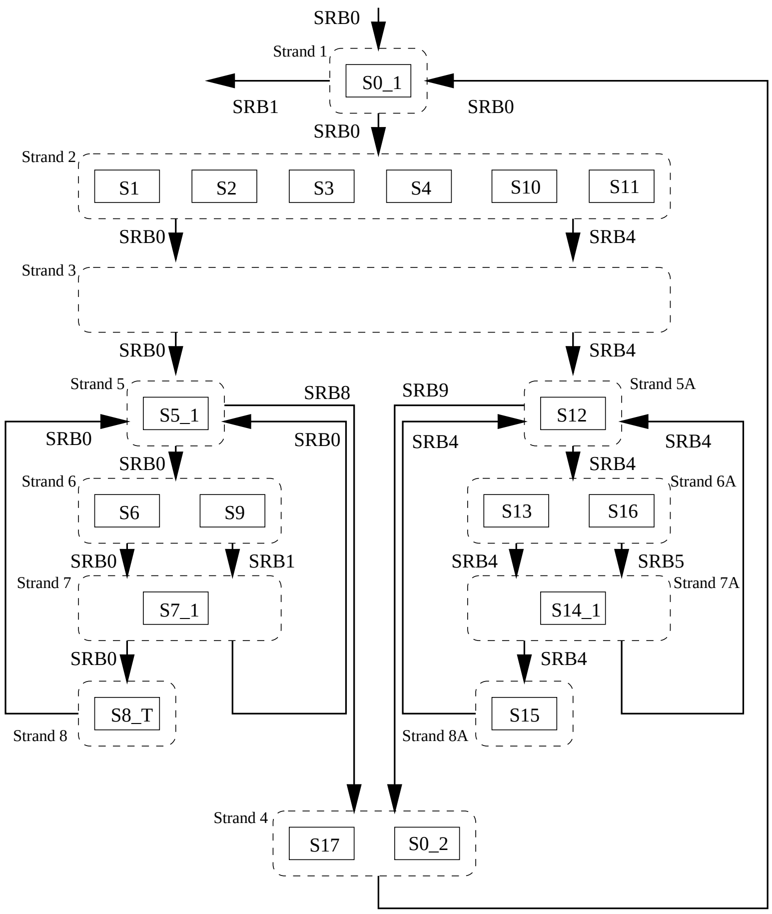


Figure 9: RS: Register Allocation example

A How RS Works — A Case Study

Following the example first presented in section 4, figure 6 shows the strand partitioning. The code generator performs an allocation of the synchronization events to the physical synchronization registers — i.e. the SRBs available in the RS. Such an allocation scheme is shown in figure 9 (Note that the second while loop has been depicted as well). Strands are numbered from *strand 0 - strand 7* while the synchronization registers are numbered from *SRB0 - SRB7*. Solid arcs represent a *SRB* associated with a synchronization event.

For example, *strand 2* will be fired as soon as *strand 1* sets *SRB0*.

In the following figures (10, 11, 12, 13) we show how RS works using the example as a case study. We make the assumption for this example that dependence successor information has been already prefetched into the *dependence successor table*, and the execution begins from the

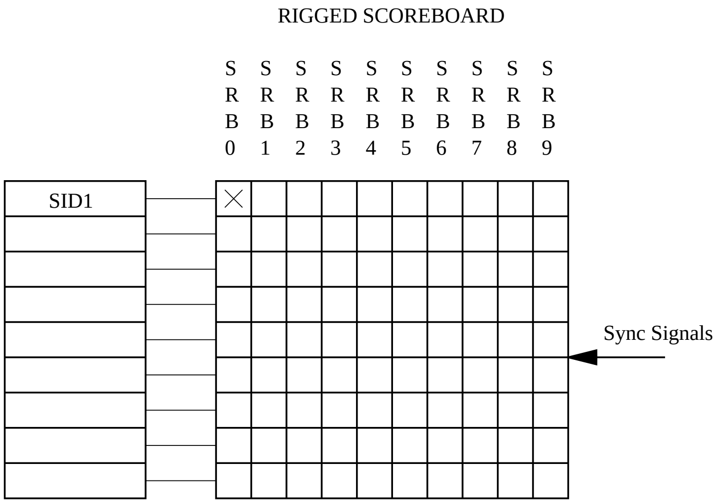


Figure 10: RS example strand 1

enabling of strand 1 (ST1):

- As shown in figure 10, the first row of the scoreboard has an entry for *strand 1* marked with *SID1*, -the strand containing the "FOR" predicate. As soon as *SRB0* is set by a predecessor strand, the firing rule is satisfied and the row is cleared after loading the strand starting address into the *ready queue*. The dependence successor to strand 1, which is strand 2, is loaded into the scoreboard. Since no further entries are marked in the scoreboard, *SRB0* is cleared.

- Figure 11 shows the scoreboard entry for strand 2. The firing mechanism is as for strand 1. Strand 1 will set SRB0, thus enabling strand 2. Strand 2 will put its starting address in the ready queue, clear its entry in the table, clear SRB0 for reuse and load strand 3 into the scoreboard.

- In figure 12, the scoreboard depicts the firing rule for strand 3. Strand 3 is activated by two non blocking split phase loads originated in strand 2. Each of these split phase loads will *sync* to SRB0 and SRB4 respectively upon completion. With SRB0 and SRB4 set, strand 3 will fire. After loading strand's 3 address into the *ready queue* and clearing it's entry in the table, two dependence successors, namely strand 5 and strand 5A will be loaded into the scoreboard. SRB0 and SRB4 are reseted.

- Figure 13 shows the scoreboard with the entries for strand 5 and strand 5A. These are the predicates for the respective while loops. Each of them is activated respectively by SRB0 and SRB4. The dependence successors to strand 5 and strand 5A are loaded into the scoreboard thereafter.

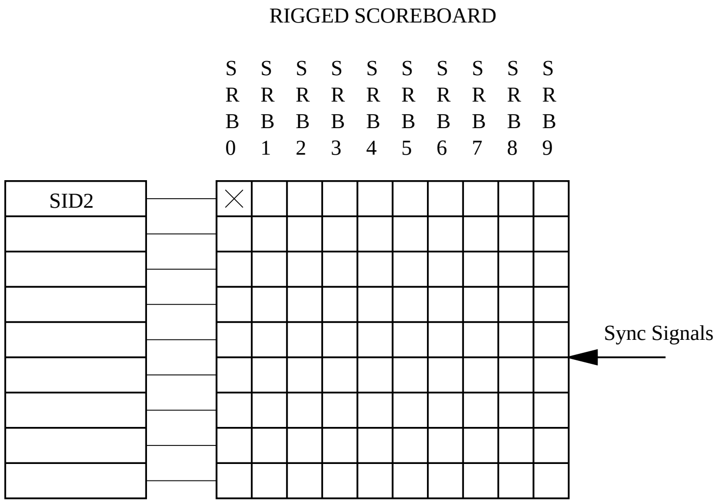


Figure 11: RS example strand 2

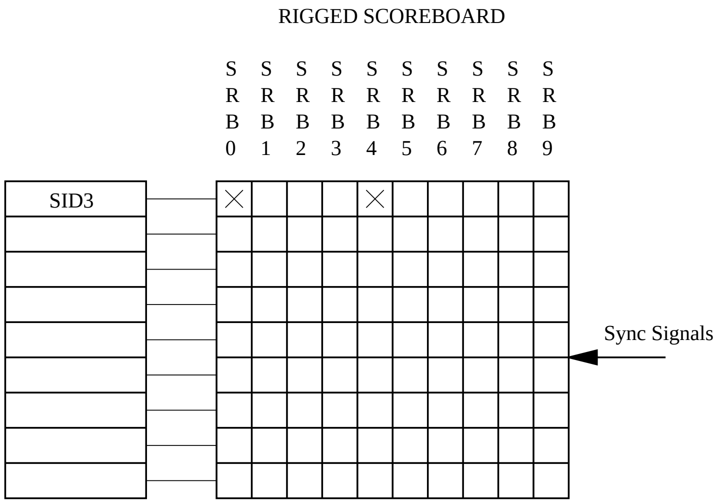


Figure 12: RS example ST3

- An interesting case is depicted with figure 14. Both, strands, 5 and 5A, are implemented as *CONDSYNC*. According to the outcomes of the predicates (evaluated by the above two strands) they will fire either another iteration or exit the corresponding loop (recall that they are in two while loops respectively). The dependence successors for strand 5 are strand 6 and strand 4 while the dependence successors for strand 5A are strand 6A and strand 4. When the result of the predicate evaluation in strand 5 indicates the execution of another iteration Strand 6 will be enabled with SRB0 set. On the other hand, if the

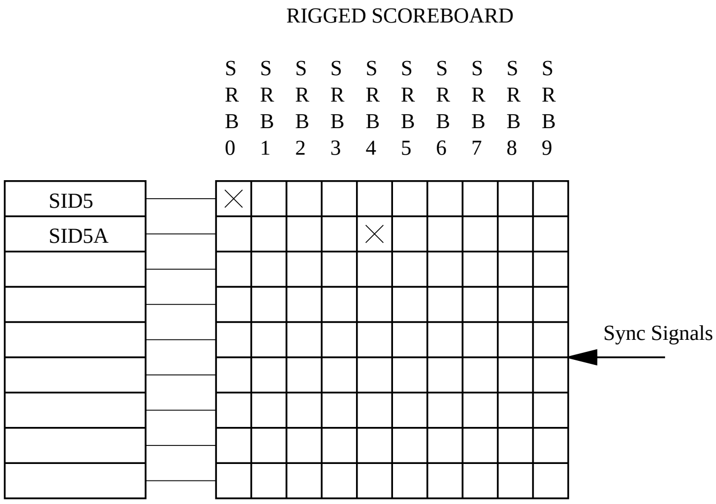


Figure 13: RS example strands 5 and 5A

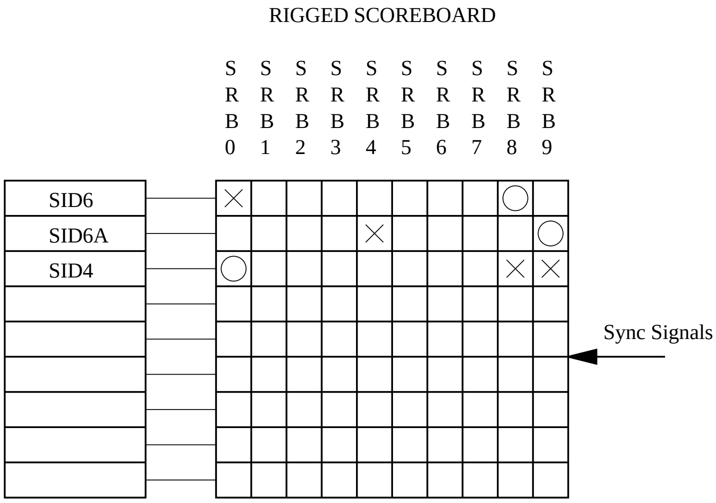


Figure 14: RS example strands 6, 6A and 4

predicate evaluation indicates the need to exit the loop, then SRB8 will be set and this will remove strand 6 from the matrix without firing it. Analogously, strand 6A will be enabled with SRB4 and removed with SRB9. Note that strand 4 is a dependence successor for both strand 5 and strand 5A. However, it only appears once in the dependence successor table and the matrix.