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

A Superstrand Architecture and its Compilation

Andrés Márquez
Kevin B. Theobald
Xinan Tang
Thomas Sterling †
Guang R. Gao

CAPSL Technical Memo 18
March 14, 1998

†Jet Propulsion Laboratory, 4800 Oak Grove Dr., Pasadena, California 91109

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 report, we present the *Superstrand* architecture and its underlying execution model. This architecture exploits the notion of a *strand* – a block of instructions grouped together as a scheduling quantum of execution. A strand is enabled at runtime only when all necessary data and control dependences are satisfied. Operations (e.g., memory operations or branches) which may incur long and/or unpredictable latencies will be made *split-phase*: an instruction which initiates such an operation and the instructions which may depend on the results will always be placed in different strands. The Superstrand execution model gives the hardware access to a pool of strands resulting in a large "window" of instructions, necessary for sufficient instruction-level parallelism, while reducing hardware complexity and the penalties associated with conditional branches and load stalls.

The main contribution of this work are:

(1) a proposed Superstrand program execution model

(2) a compiler framework under which programs can be effectively partitioned into strands

(3) a proposed Superstrand architecture which includes a novel hardware strand synchronization unit implementing fine-grain synchronization between strands by utilizing explicit inter-strand dependence information discovered by the compiler

(4) an implementation of the proposed compiler framework and the architecture (through simulation), and a report of the experimental results.

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, while maintaining the sequential semantics of a single program counter. The dominant trend today is to use increasingly powerful hardware to detect instruction-level parallelism (ILP) dynamically. Such hardware looks at a window of instructions and tries to find mutually-independent instructions within that window. Yet the performance gains so far have been only modest. There is an open debate today whether such techniques are capable of achieving issue rates significantly higher than 4–8 instructions per cycle, and, even if they can, whether the hardware required to achieve this could be put to better use elsewhere.

Many techniques have been developed to increase the number of independent instructions which can be found within the window, such as *register scoreboarding* [35], *out-of-order execution* (OOO) [18], *register renaming* [18], *branch prediction* [20], *dependence speculation* [31], and *data speculation* [24]. However, several factors can limit the exploitation of ILP from a single window:

- Load stalls are a source of highly unpredictable latencies. Compilers cannot predict most cache misses with certainty. It is usually not possible to fill the gap between a load that misses the cache and the instructions that use the load value with enough independent instructions to prevent the processor from wasting cycles.
- In spite of the sophistication (and hardware costs) of state-of-the-art branch prediction techniques, there will always be miss-predictions, which will become increasingly expensive as both execution pipeline depths and issue widths increase.
- Dynamic reordering and speculation hardware complicate matters for compiler optimization. It is not yet clear how best to utilize OOO dynamic scheduling mechanisms and register renaming at compile time (although some have tried [23, 30] but with limited success). It is also not 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, two-level branch prediction schemes are expected to be widely used in future microprocessors, yet we know of no compiler which can model this in its code analysis.

Furthermore, studies based on sequential program traces have shown that a large window is required for significant parallelism [3, 36]. 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).

Traditionally, multithreading [1, 2, 5, 11, 15, 17, 21, 25] has been proposed as a technique for tolerating the long latencies of certain operations by allowing alternate threads of computation to execute while waiting for these operations to finish. Multithreading has been advocated for multiprocessors, where the long-latency operations in question are remote memory references.

Recently, however, multithreading has been used as a viable solution to hide short-latencies at the architecture level [9, 31, 37, 38]. We believe that a *suitably-designed* multithreaded program execution model can serve as the basis for a multiple-issue uniprocessor, and that such a processor can achieve reasonable levels of parallelism without the need for expensive hardware mechanisms to support the aggressive superscalar techniques mentioned above.

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 compiler to become a scheduling quantum of execution. Operations (e.g., memory operations or branches) which may incur long and/or unpredictable latencies will be made *split-phase*: An instruction which initiates such an operation and the instructions that may depend on the result of the operations will always be placed in different strands. As a result, a strand can be executed in a *non-preemptive* fashion: once a strand begins execution, its context is not removed or swapped out of the active register area until execution has completed. The enabling and scheduling of strand execution follow the following principle: a strand becomes enabled (ready) and can be scheduled for execution only when all necessary dependences (both data and control) have been satisfied. All strands whose data and control dependences have been satisfied are eligible for execution, and are placed in a "ready queue." When the execution pipeline finishes executing a strand, it takes the next strand from the ready queue.

Our restrictions on strands remove both control dependences and long-latency data dependences from the execution pipeline. Thus, hardware to ameliorate the effects of these dependences is no longer needed. Instead, these dependences are handled by a separate, relatively simple piece of hardware. As long as there is enough parallelism in the program so that at least one strand is in the ready queue when a strand terminates, the new strand can be fed into the pipeline without any bubbles.

The main results of this report include:

- **Program Execution Model:** The Superstrand program execution model is proposed and presented (see Section 2).
- **Compiler:** A compiler framework for the Superstrand architecture is proposed. A method of strand partitioning is developed and presented (see Section 3).
- **Architecture features:** The Superstrand architecture model is proposed and presented. 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 register bits" (called SRB's) 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 (see Section 4).

- **Implementation and Experiments:** We recognize that a significant number of experiments remain to be performed before any conclusion can be made to a new architecture such as the Superstrand architecture, which is based on (yet-to-be-established) ideas. However, we feel that the preliminary results below should be a valuable source for discussion:
  - The proposed strand partitioning method has been implemented in our EARTH-C compiler framework [14, 32]. We present the experimental results of compiling six *Olden* benchmarks [4, 26] and show the profiles of strands in these program examples (see Section 5.1).
  - A version of the proposed Superstrand architecture has been implemented in our execution-based simulation testbed SEMi ([34]). We report the benefits of the Superstrand architecture in eliminating branch penalties and drastically reducing memory access stalls (see Section 5.2).

2 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, and executed.

2.1 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*, has its own context called a *frame*, which is a block of memory holding function parameters and local variables. As in some other multithreaded models [7, 15], frames are dynamically allocated from a heap, and are typically linked to one another in a non-linear structure (such as a tree), rather than being confined to a linear stack. This allows threads to run concurrently, even if they are instances of the same procedure. Threads can be created and destroyed at run-time, and threads can create new threads.

Code at the level of a function instance still requires long-latency fetches and conditional branches, which impose heavy costs on superscalar processors. Multiple threads can be interleaved to hide the latencies, but it is not always easy to find "course-grain" parallelism in the form of parallel function calls. Furthermore, switching from one thread to another imposes context switching costs.

Instead, a Superstrand processor divides a threaded function into smaller units called *strands*. A strand is a sequence of instructions which, once activated, should be executed to completion without any stalls due to control dependences or long-latency operations. Strands are constrained by an "enabling rule" which states that a strand will not start executing until all data needed by that strand have been produced (except for intermediate results produced *within* the strand itself) and all control dependencies are satisfied. How this rule is enforced is explained in the next subsection.

Strands, therefore, can be scheduled atomically, for, once a strand is ready to begin execution, nothing can cause it to stall. The enabling rule guarantees that hardware no longer needs to check dynamically for dependences between instructions in different strands. This can reduce hardware complexity considerably. Rather than trying to schedule many instructions from a large window with many potential dependences to check, the processor can instead execute several strands concurrently, knowing that it only needs to check for dependences within each strand's instruction window. Furthermore, branch penalties and load stalls within strands are eliminated.

Not any random group of instructions is suitable for a strand. First, there can be no conditional branches. Therefore, only instructions in the same "basic region," which can be considerably larger than a basic block(see Section 3), may be grouped into one strand. Control flow is handled *outside* of the strand level, using a special conditional operation described in Section 2.2. Secondly, a load instruction and any instruction which depends on the load cannot be in the same strand, unless it is known that the load delay can be handled without incurring stalls (either because the compiler knows the object will be in the cache [22] or because enough instructions can be inserted between the load and its first use to cover maximum memory latencies).

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 *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. Thus, a strand's total context consists of the private register pad, plus several other pointers: a *thread register pointer*, which points to the base of the register set allocated to the thread and shared among all strands in this thread; a *frame pointer*, which points to the frame corresponding to the thread containing the strand; and an *instruction pointer*, the address of the next instruction to be executed. Since the register pad itself is small and does not need to be initialized, the cost of context switching from a completed strand to a new strand in the same thread is very low.

2.2 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, the compiler or programmer must explicitly identify data and control dependences between this strand and others. Each strand has a dynamic *synchronization count* (or *sync count*), which counts the number of dependences on which this strand depends but are currently unsatisfied. If another strand produces a required datum or control condition, the producer performs a *sync operation*, which signals the sync slot that one more dependence for this strand has been satisfied. The CPU then decrements the strand's sync count. If the sync count reaches 0, all dependences have been resolved, and the strand becomes eligible for execution. This simple mechanism has its origins in the static dataflow model of computation [8], and was further extended in the EARTH multithreaded execution model [15].

Conditional branches are avoided by providing special *conditional* sync operations. A conditional sync operation allows the code to send a sync signal to one or more sync slots *conditionally*, according to the value of an expression (see Section 4). Long-latency memory references are tolerated through the use of special *split-phase* loads and *posted* writes.

As soon as a strand becomes eligible for execution, it is placed in a *ready queue* to await execution. If the CPU has resources to execute this strand two scenarios are possible:

- In case the previous strand which used the same resource is part of the same thread the CPU executes the strand immediately.
- Otherwise, the CPU first loads the context pointers and then executes the strand.

Since the strand is non-preemptive, it stays in the CPU until it finishes, at which point it leaves the CPU, the sync count for this strand is reset to the original number of required dependences, and the CPU fetches a new strand from the queue.

It is feasible to determine when a strand is nearing completion since the unpredictability of strands latencies is strongly reduced. At this point, the fetch unit can begin prefetching the instructions in the next strand in the ready queue while the current strand is finishing. As long as there is enough parallelism so that there is at least one strand in the ready queue whenever the CPU finishes a strand, long-latency operations and conditional operations can overlap the execution of other strands. Thus, it is not the case that moving conditional dependences and long-latency operations outside of strands merely moves the associated penalties to different times.

The CPU can hold a limited number of strands in the active state, depending on its hardware resources. When a strand terminates and leaves the CPU, the CPU goes to the ready queue for another strand. A *strand scheduling policy* determines the order in which strands are activated.

A simple FIFO scheduler would activate strands in the order in which they become enabled, although other scheduling policies can be used to try to optimize strand ordering for locality.

Overall, when a threaded function is called, a thread and an associated frame are created, parameters to the function are placed in the frame, and an initial set of strands specified by the program is placed into the "ready queue." These *initialization strands* are automatically enabled because they will only use the already available function parameters. All remaining strands are waiting for their sync counts to reach 0. The initialization strands will send sync signals to other strands, enabling some of them in the process.

3 Program Representation and Strand Partition

In this section, we will outline a program representation, called the *program dependence flow graph* (PDFG), and briefly describe the strand partitioning strategies for these graphs.

A *Program Dependence Flow Graph* (PDFG) is an executable program representation, in which the control structure and data dependences of a program are represented in a hierarchical fashion. A PDFG can be easily built based on the *Program Structure Tree* (PST) representation. A PST of a program is a hierarchical representation of the control structure of the program [19]. A PDFG is a PST, modified and augmented with additional nodes and edges to represent precedence constraints caused by data dependency and control transfer.

We illustrate our program representation with an example. Figure 1 is a program that counts the number of times a symbol appears in two linked lists. During each iteration of the outer loop, a symbol is retrieved from a symbol array (`Buffer[]`), and then this symbol is checked against the lists, `list_header` and `list_header1`. Whenever a match is found, the symbol's base weight `w` is added to a partial sum (`r` or `r1`). After the lists have been searched, the sum of `r` and `r1` is put in the weight array (`Weight`).

The PST of the example program is shown in Figure 2, 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}), containing 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 [19]) such that the nodes in the same basic region have the same parent in the PST with the identical control dependence. In Figure 2, dash-edged rectangles outline these basic regions. For example, the compound *while-loop* statement S_5 contains one subcomponent (loop body) which is a basic region, consisting of three nodes: S_6, S_7, and S_9. Furthermore, they share the same parent node S_5, on which all three nodes are control dependent.

Now the PST is modified to create the PDFG. First, we introduce data dependence edges between nodes within the same basic region, i.e., we build a *Data Dependence Graph* (DDG)

```
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 1: C Code Computing Symbol Weights

for each basic region. In Figure 3, 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 after introducing the control edges in the next step, a region node (corresponding to a compound statement) is split into a pair of nodes: a "begin" and a "end" node as illustrated in Figure 3.

Next, 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 3, 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 (S_7 in the original program listing). It has one input control edge from the region-begin node of S_7, and

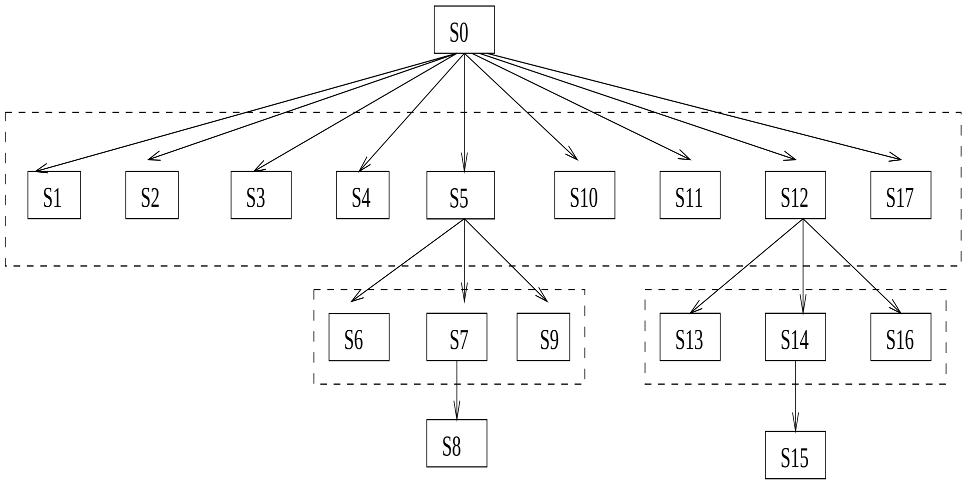


Figure 2: The PST of the Example C Code

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 directly connected to the merge node X (since the "else arm" is empty). The output of X goes back to the region-end of S_7.

In summary, a PDFG consists of two parts: a set of DDGs built for each basic region, and a CFG connecting nested regions. Intuitively, there is no DDG edge crossing regions, and control flow enters a SESE region at the beginning and leaves at the end. As we showed earlier, a PDFG of program P can be built from its corresponding PST.

3.1 Operational Semantics of a PDFG

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).
- 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 any one of its input signal arrives. The firing of an enabled merge node will simply send a signal to its output arc.

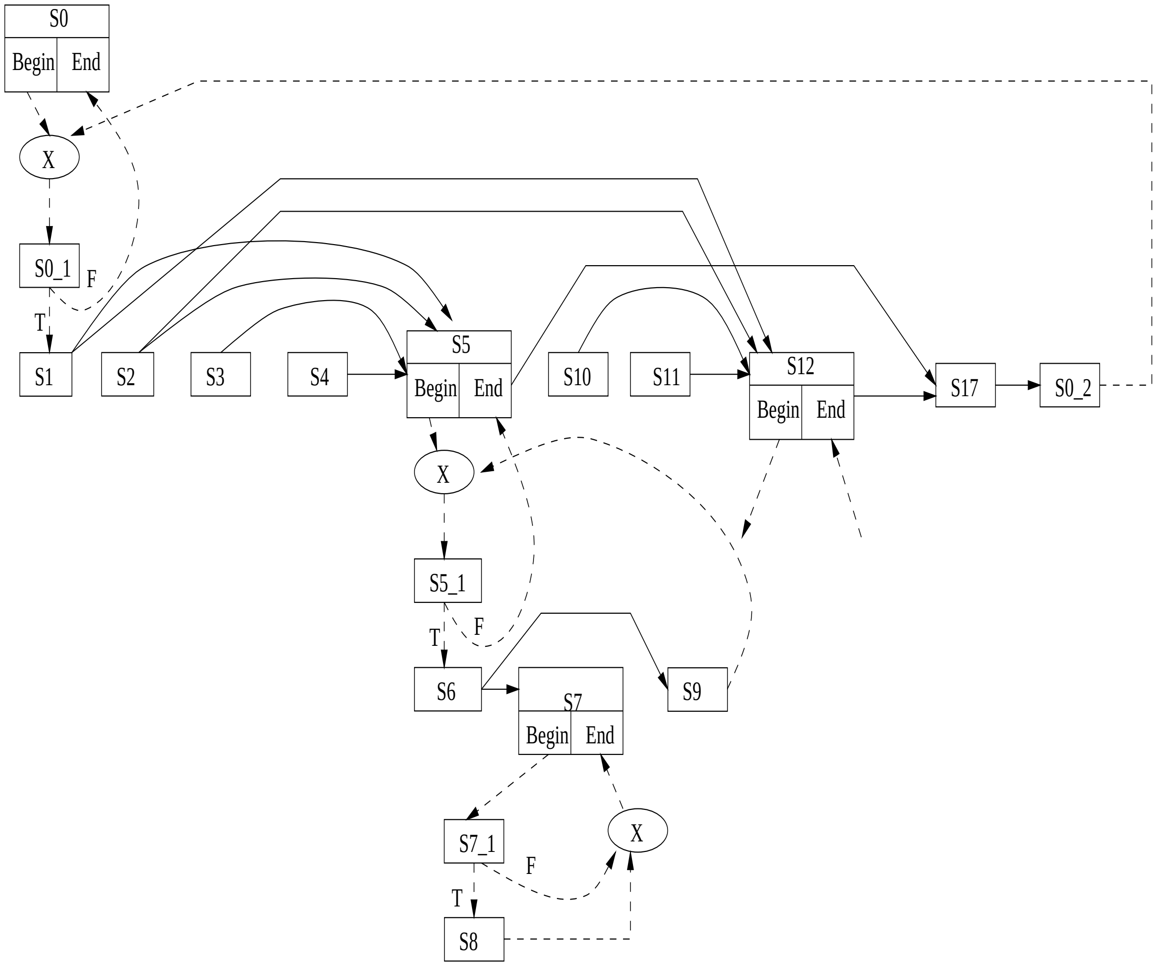


Figure 3: Program Flow Dependence Graph

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.

3.2 Partitioned PDFG and Stranded Code

Next, we outline how nodes in the PDFG can be grouped into strands. One of the key problems in strand partitioning is to decide which type 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 that must be cut:

1. Remote DDG edges: DDG edges starting at nodes that have long latency operations. These nodes are called *remote* nodes and will be executed in a *split-phase* manner. For

example, nodes that access *arrays* and/or *heaps* can be *remote*. In Figure 3 node S_1 and S_2 are remote since they contain either array or heap accesses.

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). We must cut this edge so that node S_{5_1} and S_6 can be put in two different strands.

Figure 4 illustrates strand partitioning for the program listed in Figure 3. A strand is delimited by a dashed rectangle 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. The nodes and the edges incident on them are omitted for clarity. In Figure 4, all remote nodes are shaded and are separated from the nodes which depend on them. For example, S_1 and S_2 are in strand 2 while S_5 and S_{12}, which depend on S_1 and S_2, are in strand 3.

The hierarchical representation greatly facilitates 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 the strand partitioning algorithms is beyond the scope of this paper. We simply show the partitioning results in Figure 4.

In Figure 4, we use circled numbers to represent how many remote DDG edges exist between two strands, since only nodes connected by remote edges need synchronization. For example, there are two remote DDG edges from strand 2 to strand 3. They represent the union of the original DDG edges (S_1, S_5),(S_2, S_5) and (S_1, S_{12}), (S_2, S_{12}).

In Figure 4, some optimizations are applied to improve the partition. For example, S_7 and S_{7_1} are merged into one strand since there is a single edge connecting them, and this edge does nothing but start 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 5 lists the corresponding *stranded* code and Table 1 lists synchronization signals among strands. Please note 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 heuristics can be used to leave $CondSync$ at the end of a strand so that the strand switching must happen after this operation is issued.

The first two columns of Table 1 are 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.

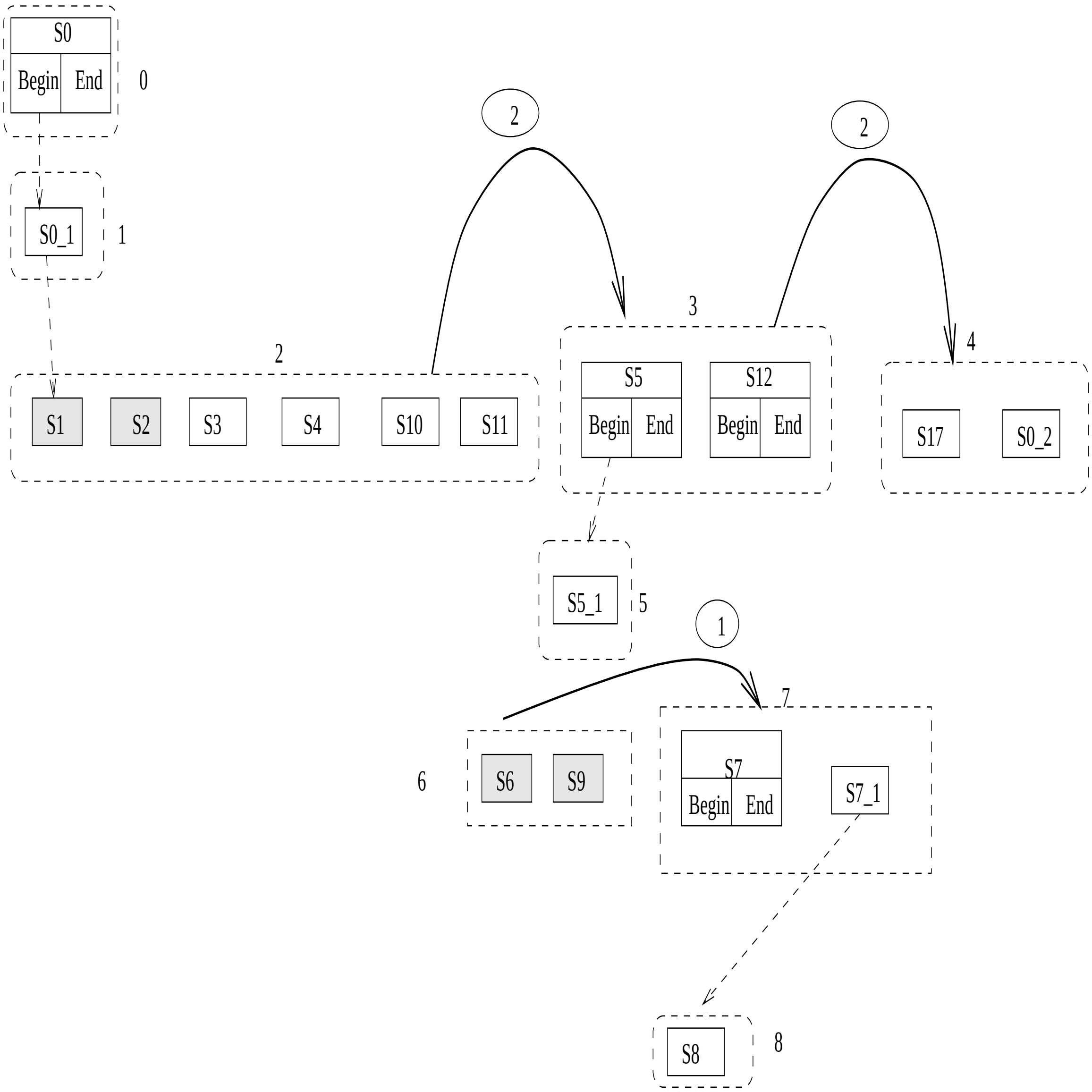


Figure 4: Partitioned Strands

Even though the two types of signals actually coexist, representing two different dependences, *control and data* dependences, the underlying hardware will treat them in the same way as the dataflow dependence. Thus, the dataflow firing rules introduced earlier can be applied uniformly to govern strand synchronization.

```
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:
S17:  Weight[i] = r + r1 ;
S0_2: i = i + 1;
      END_STRAND();

STRAND_9:
```

Figure 5: Threaded C Code With Explicit Synchronization

4 Efficient Architecture Support for Strand Synchronization

A key element separating the Superstrand architecture from superscalar is the maintenance and use of "explicit dependence" information in supporting the synchronization and scheduling of strands according to the Superstrand execution model in Sections 2 and 3. Section 2.2 describes a system where individual strands are associated with sync slots containing counters. Such counters are used in some other multithreaded systems [15].

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

Systems based on actual counters have faced resource limitations. A large application can have many sync slots, which leads to two problems. First, it may be difficult to keep so many counters close to the processor, and this may increase the time it takes for a sync signal to enable a strand and place it in the ready queue. Second, a large number of sync slots means strand identifiers require many bits.

However, if we view sync counts as precious resources, which are to be allocated carefully, this leads to an elegant solution: place sync slots with short lifetimes in fast registers, and leave the rest in memory. We have developed an efficient design for implementing fast sync slots in hardware, which we call *Rigged Scoreboarding* (RS). We use this name because we use explicit dependence information discovered by the compiler, whereas conventional register scoreboarding uses implicit dependence information in the sequential code discovered and resolved at runtime.

The general scheme is shown in Figure 6. At the top of the rigged scoreboard is a row of bits called *Synchronization Register Bits* (SRB's). Each bit can be set by a sync instruction executed by a strand. 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).

Each entry in the matrix is either $\times$, $\bigcirc$, or blank. An $\times$ in row i, column j means that SRB_j must be set before the strand corresponding to row i can be enabled. If a row has multiple $\times$ settings, all corresponding SRB bits must be synchronized before the strand is enabled. (The $\bigcirc$ is explained later.)

In theory, there could be one row for every strand in the program, and one SRB for every synchronization event in the code. Since this would make the matrix extremely large and sparse, both rows and columns should be reused like data registers. For instance, only the

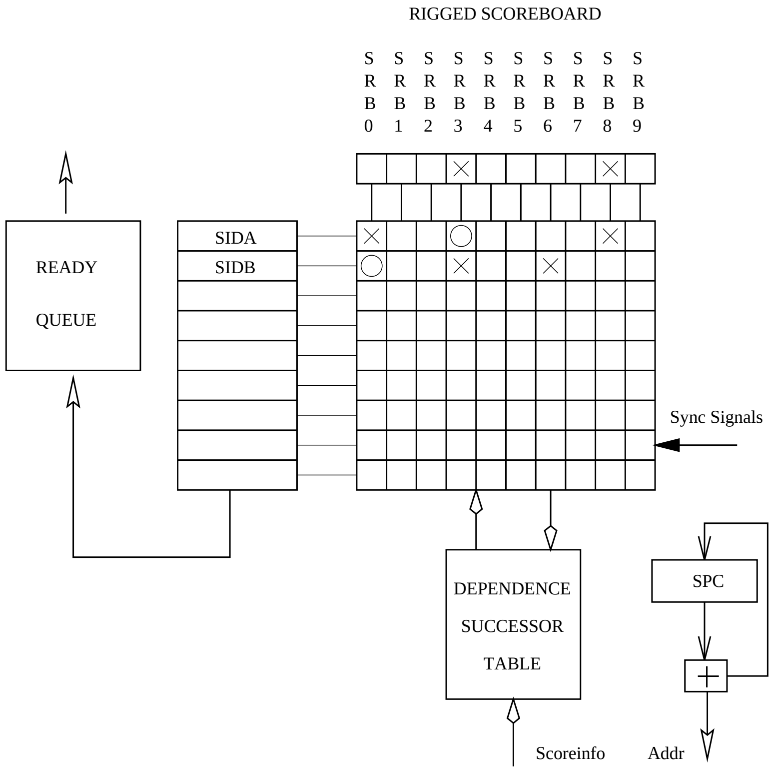


Figure 6: Rigged Scoreboarding (RS)

rows corresponding to strands which are dependent on currently-active strands need be in the matrix. No other strands can receive a sync signal from a currently-active strand. The other strands remain in a *dependence successor table*, which contains the firing rules for all strands and a list of potential successors of each strand. Similarly, since a synchronization event is only "live" from the time a *sync* instruction is executed to the time the strand is enabled and placed in the ready queue, the compiler may map several individual synchronization events to one SRB using a graph coloring algorithm based on interference graphs, much like a register allocator.

Whenever a sync instruction is executed, the specified SRB is set, and the rows of the matrix are associatively searched for any active row in which every $\times$ entry corresponds to an SRB which is on. For each row satisfying this condition, the corresponding strand is put in the ready queue, and the strand's successors are fetched from the successor table. Each of it's successors is put in the matrix, along with the $\times$ and $\bigcirc$ entries representing the firing rules, unless it is already there. The enabled strand is removed from the matrix unless it is in its own successor list (which means it can send a sync to itself).

In most cases, the SRB's which caused the enabling can be cleared, since they have served their function of synchronizing the strand. However, 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 "fan-out" of two or more. (It is this ability to share a synchronization bit among multiple strands that makes RS more attractive than the counter scheme described earlier.) Therefore, bit SRB_j can't be cleared until all rows with $\times$ in column j have been removed.

Conditionals are handled with a special conditional sync instruction, which compares two registers and sends a sync signal to one specified SRB or the other, depending on the comparison. It is usually the case that the two SRB's sync different strands. When the strand containing the conditional sync is enabled, both successor strands must be put in the matrix, but only one will be enabled. The $\bigcirc$ state is used to delete the other entry from the matrix. When SRB_j is set, all rows with a $\bigcirc$ in column j are deleted.

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

1. When a strand executes a *sync* instruction specifying SRB_k, set that SRB.
2. For row i, if any column j that is $\bigcirc$ is matched by a set bit in SRB_j
   (a) remove row i.
3. For row i, if all columns j that are $\times$ are matched by a set bit in SRB_j:
   (a) Fetch the strand ID in row i, and send it to the ready queue.
   (b) Remove row i.
   (c) Look up the strand's successors in the dependence successor table. For each successor, allocate a row in the matrix, and copy the $\times$ and $\bigcirc$ entries into the row.[1]
4. For each column k, if there are no remaining $\times$ symbols in that column, reset SRB_k.

With this basic design, strands from only one thread may be in the matrix at one time. This is because sync instructions are coded for specific SRB's, but if two threads try to share the RS, they may try to use the same SRB's. If one wants the processor to be able to handle strands from different threads simultaneously, then some form of sharing, analogous to multiple register windows, is needed, such as multiple SRB sets or SRB renaming.

Note that, with enough entries in the matrix, the RS will never deadlock. The code generator must know the dimensions of the matrix in order to prevent possible overflows of the matrix. If SRB allocation determines that too many rows could be allocated at one time, the compiler must reduce parallelism by combining strands.

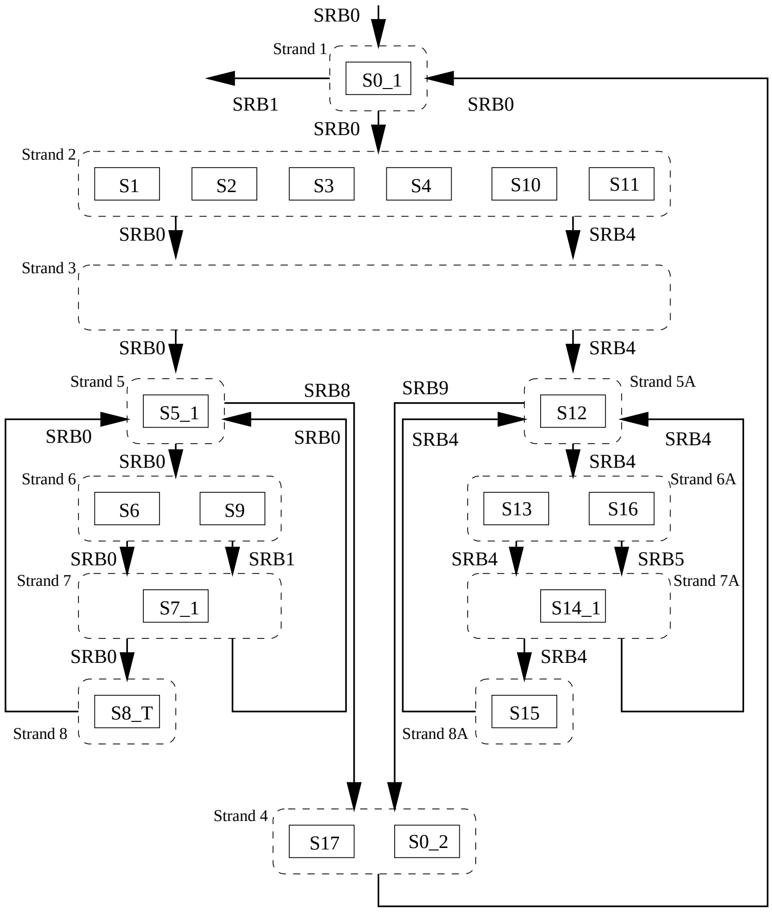


Figure 7: RS: Register Allocation Example

4.1 How RS Works — A Case Study

Figure 4 shows the strand partition of the example first presented in Section 3. The code generator performs an allocation of the synchronization events to the physical synchronization registers — i.e., the SRB's available in the RS. Such an allocation scheme is shown in Figure 7 (note that the second while loop is depicted as well). Strands are numbered from 0–7 while the synchronization registers are numbered from SRB0–SRB7. Solid arcs represent an SRB associated with a synchronization event.

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

In Figures (8, 9, 10, 11) 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

[1]This operation must finish before the enabled strand has started executing. Hardware design must take this into account.

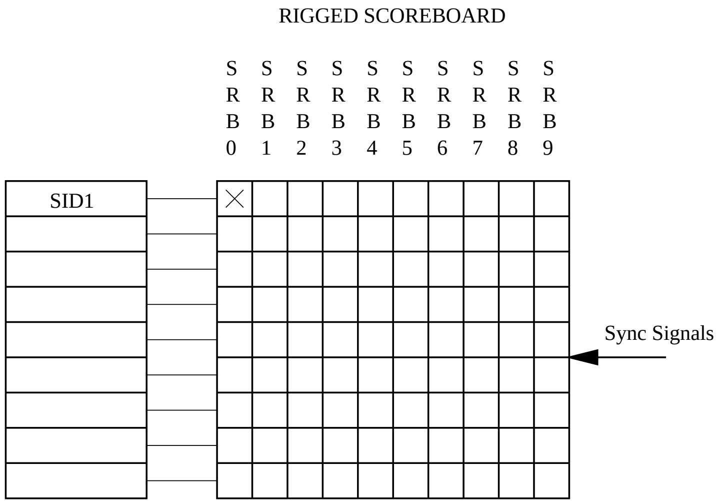


Figure 8: RS Example Strand 1

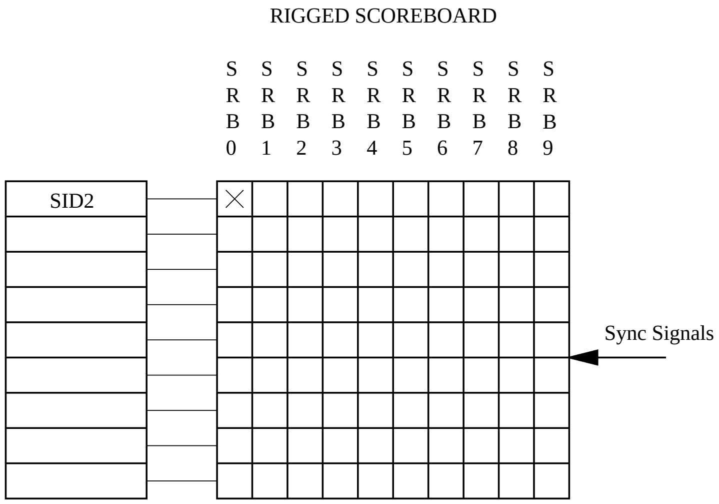


Figure 9: RS Example Strand 2

prefetched into the dependence successor table, and the execution begins with the enabling of strand 1 (ST1):

- As shown in Figure 8, 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,

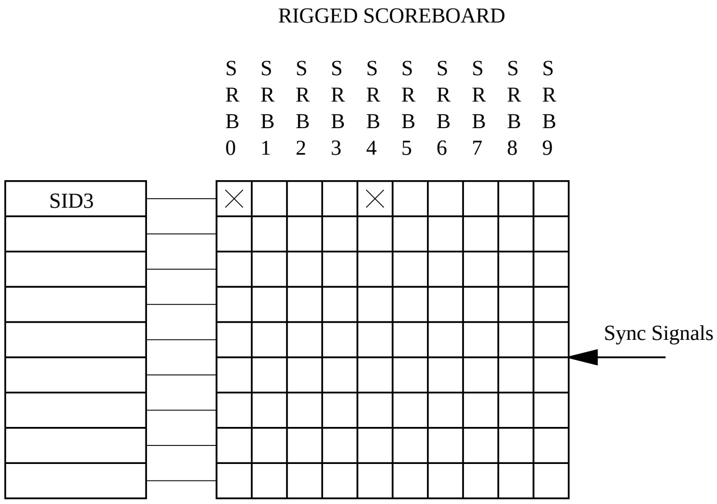


Figure 10: RS example ST3

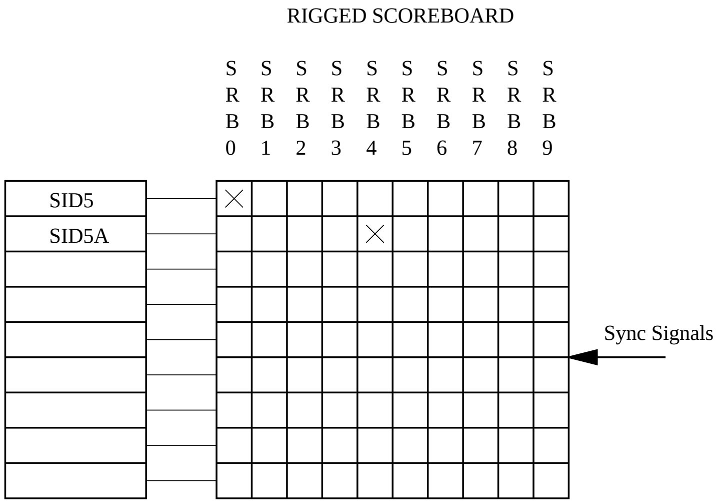


Figure 11: RS Example Strands 5 and 5A

which is strand 2, is loaded into the scoreboard. Since no further entries are marked in the scoreboard, SRB0 is cleared.

- Figure 9 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.

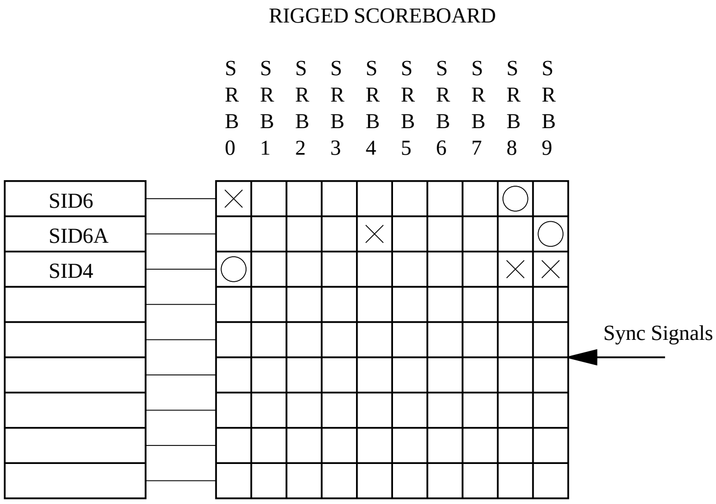


Figure 12: RS Example Strands 6, 6A and 4

- In Figure 10, 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 its entry in the table, two dependence successors, namely strands 5 and 5A will be loaded into the scoreboard. SRB0 and SRB4 are reset.

- Figure 11 shows the scoreboard with the entries for strands 5 and 5A. These are the predicates for the respective while loops. Each of them is activated respectively by SRB0 and SRB4. The dependence successors to strands 5 and 5A are loaded into the scoreboard thereafter.

- An interesting case is depicted with Figure 12. 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 strands 6 and 4 while the dependence successors for strand 5A are strands 6A and 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 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. Similarly, 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.

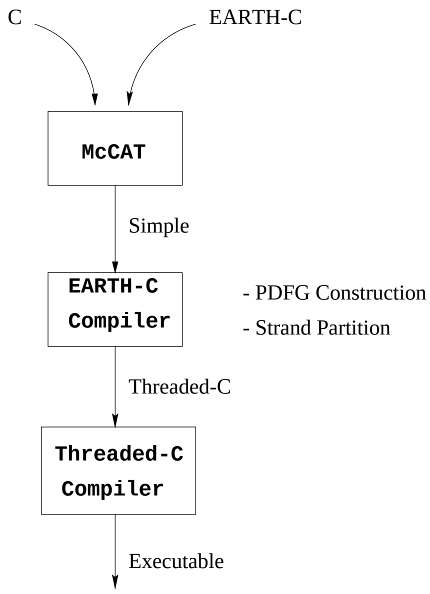


Figure 13: The EARTH-C Compiler Platform

5 Experimental Results

In this section, we present some experiments on the performance of the proposed Superstrand execution and architecture model and its compilation technology. The first set of experiments test the strand partitioning techniques given in Section 3, and present statistics about the strands generated by the compiler. The second set of experiments simulate the architecture of Sections 2 and 4 and compare the performance with a nearly equivalent sequential architecture.

5.1 Compilation Experiments

The strand partitioning strategy mentioned in Section 3 has been implemented using the EARTH-C compiler [14, 32], which is built on top of the McCAT compiler [13]. As shown in Figure 13, the EARTH-C compiler can take programs written in either parallel EARTH-C or sequential C, although the former yields more parallelism. It performs extensive dataflow analyses such as *points-to (alias)* and *heap* analysis for pointer variables, to build PDFGs. Using PDFG's, strand partitioning and generation will be applied. The output of the EARTH-C compiler is an explicitly-stranded Threaded-C program [15], which can then be compiled by a separate compiler, one augmented with a backend register/SRB allocator to produce executable code that can be fed into our superstrand architecture simulator to be described in the next section.

One advantage of the McCAT compiler is that the intermediate representation is kept in a structured form, which facilitates PDFG construction. Based on the PDFG representation, the general idea of the two-pass strand partitioning algorithm is as follows:

Name	Description	Avg. stmts per strand	Max. stmts per Strand	Memory ratio	Avg. syncs per strand
power	Power system optimization	2.37	30	0.59	0.23
tsp	Heuristic Traveling Salesman	4.80	12	0.34	0.25
em3d	3-D electromagnetic propagation	3.70	21	0.52	0.25
health	Health-care system modeling	2.25	14	0.64	0.32
mst	Bentley's min. spanning tree alg.	2.39	8	0.66	0.40
perimeter	Perimeter of raster image	2.40	24	0.49	0.33
Average		2.98	18	0.54	0.30

Table 2: Statistics by Considering Memory Effects

(1) apply a *remote-level*-based partitioning algorithm [14, 33] to each basic region and partition strands accordingly

(2) traverse the CFG (based on program regions) to cut control edges and split strands further. Our PDFG representation simplifies the implementation of this new strand partitioning algorithm greatly.

To test our compiler technology, We have written EARTH-C versions of six Olden [26, 4] benchmarks (see Table 2). These irregular benchmarks use pointers extensively and have irregular, dynamic control structures. Thus, they form an interesting benchmark suite to test our partitioning algorithm.

Table 2 lists the static properties of the stranded code generated by our compiler. The properties of interest are the average and maximum statements per strand, the "memory ratio," and the average synchronization signals sent per strand. The memory ratio is the ratio of the number of memory accesses estimated at compile time versus the total number of statements. (In this table, array accesses, structure references, and pointer dereferences are counted as memory accesses while scalar variables are assumed to be residing in either cache or a register with high probability.) In this context, "statement" refers to a statement in the internal representation used by the EARTH-C compiler. This is based on a simplified three-address format, but high-level data formats (such as arrays and structures) are retained. Thus, one statement is roughly equivalent to three instructions.

Our main results are summarized as follows:

- Our strand partitioning algorithm is practical and amenable to efficient implementation. Compared to other dataflow analyses such as *points-to* analysis, the strand generator usually takes less compiling time. Furthermore, the new partitioning algorithm can also be easily incorporated into existing DDG-based partitioning algorithms, and the PDFG has proven to be a nice tool for strand partitioning.

- The length of strand varies with applications, from long strands in *tsp* (4.80 statements per strand) to short strands in *health* (2.25 statements per stand). In terms of instruction count, on average, the number of instructions per thread is roughly 9 ($= 2.98 * 3$). The maximum strand length in *power* reaches 30 statements.

- The sync signals per strand measures the overhead of synchronization among the strands. There are two types of sync signals in our model: (1) immediate signals sent by explicit synchronization instructions, and (2) delayed signals associated with the completion of split-phase loads. The last column counts the first type of signals appearing in strands. This type of signal is usually caused by control transfer.

- The memory ratio used in Table 2 shows that the number of memory accesses versus total statements varies from a low of 34% with *tsp* to a high of 66% with *mst*. If we perform again the conversion from statements to instructions we end up with a ratio of memory accesses on the order of 11% to 33%, which is comparable to the percentages of load/store instructions in sequential programs.

5.2 Architecture Experiments

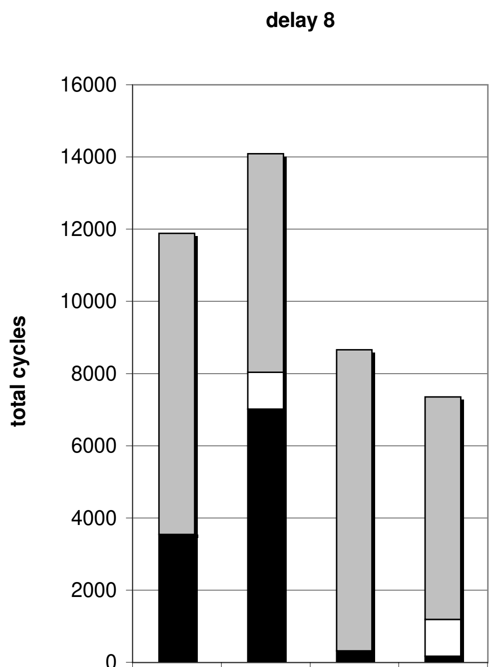
For our testbed platform we chose SEMi [34], an accurate execution-level RISC architecture simulator. The core of SEMi simulates the Intel i860XP processor [6]. However, SEMi also simulates other key aspects of a real machine, including its memory system and caches. One attractive factor of SEMi for our experiments is the fact that both correct behavior and accurate timing statistics of SEMi have been validated by comparing its results in numerous benchmarks with actual measurements conducted on a real reference machine ([12]).

We have extended the base instruction set with additional instructions for inter-strand synchronization, strand termination and split-phase loads[2]. We hand-coded our stranded example from Section 3. For comparison, we derived sequential code from the stranded code by replacing the special instructions with the equivalent native i860 instructions.[3] The input data sets were chosen to fit entirely into the cache to study the worst-case impact of stranding overhead when the benefits of load/store latency hiding are eliminated by a near-perfect hit rate.

In Figures 14 and 15 we present the total number of cycles for the execution of stranded and non-stranded code for our example using both enabled and disabled caches. As shown in the bar graphs, each cycle is classified as either a regular *instruction cycle,* a *load/store stall,* or

[2]By only extending the instruction set we still yield accurate simulation for the original instructions. A more intrusive change, as for a original superstranded machine, would have yielded a better performance for the additional instructions on the expense of sacrifying accuracy.

[3]The sequential version was first coded in C and compiled with -O4 using a commercial compiler. In some cases, the stranded version ran twice as fast, because the compiled sequential version generated more loads than the stranded version.



	strand no cache	seq no cache	strand with cache	seq with cache
instruction cycles	8342	6055	8342	6155
branch stalls	0	1021	0	1021
load/store stalls	3545	7018	319	171

Figure 14: Total Execution Times for 8/10 Cycles R/W Memory Access

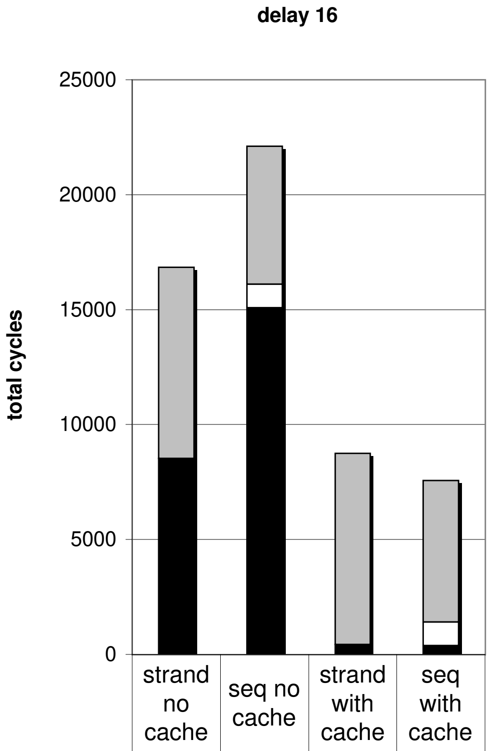
a *branch stall.* Instruction cycles include i860-dependent delays.[4] Branch stalls include explicit bubbles due to delay slots being canceled by conditional branches, and wasted cycles where a delay slot had to be filled with a nop. For sequential code, "load/store stalls" are cycles lost because an instruction is dependent on a load. For stranded code, "load/store stalls" are cycles lost because no strand is executing and there is nothing in the ready queue. Figure 14 assumes a miss penalty of 8 cycles and a write time of 10 cycles.[5] In Figure 15, we doubled the read/write memory latencies to 16/18 cycles.

The following summarizes the results of our experiments:

Performance: The superstrand architecture does surprisingly well: even with a single execution unit (i.e., strands are executed one at a time), the stranded code runs faster than the sequential (unstranded) code when the cache is disabled. In Figure 14, we observe that the

[4]Architecture-dependent delays are inherent in the execution of specific sequences of i860 instructions where pipelining or forwarding is not possible.

[5]These memory delays are in accordance with the MANNA reference machine for normal memory accesses.



	strand no cache	seq no cache	strand with cache	seq with cache
instruction cycles	8322	5989	8322	6152
branch stalls	0	1021	0	1021
load/store stalls	8528	15093	429	388

Figure 15: Total Execution Times for 16/18 Cycles R/W Memory Access

total cycle count of the sequential code with cache disabled (14094) is 18% slower than that of the stranded code (11887). Figure 14 shows the susceptibility of the sequential code to memory latencies as opposed to the stranded version. When caches are disabled, the sequential version of the code suffers a performance drop of nearly 50%, much higher than the stranded code!

Robustness: Our experiments also demonstrate the performance robustness of the superstrand architecture in the presence of higher memory access latencies. The bar graph in Figure 15 indicates that the execution time penalty due to cache disabling is much more severe for the sequential version than the stranded version; the sequential version jumps three fold while the stranded code only jumps two fold. More importantly, the performance drop increases much faster for the sequential version than the stranded version when the memory access latency increases.

Latency Hiding: The observed performance advantage of the superstrand architecture is due to its ability to exploit fine-grain parallelism in the code very effectively, avoiding stalls due to branches and long-latency memory operations. The performance data shown in Figure

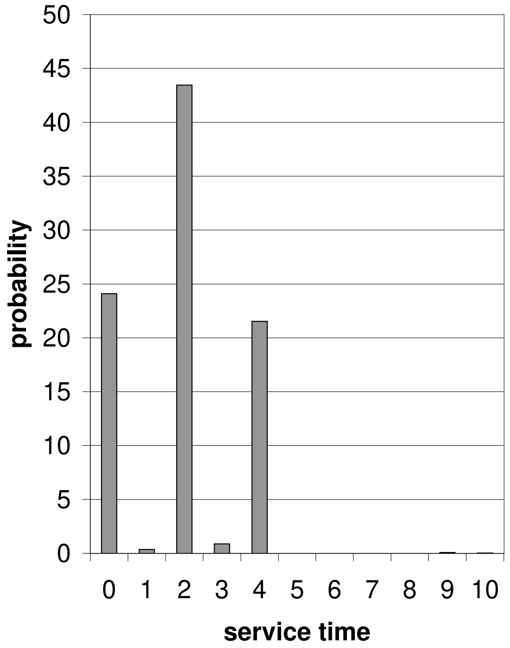


Figure 16: Ready Queue Availability Times

14 provides strong evidence that the branch and memory stalls have been tolerated effectively by our architecture. Here, we provide additional analysis based on simulation data gathered at the microarchitecture level.

In Figure 16, we show the dynamic distribution of ready queue availability times. Whenever the CPU tries to fetch a new strand from the ready queue (after a strand terminates), the simulator counts the number of cycles the new strand was waiting in the queue (if one is there). For example, a value of 43% for 2 cycles means that 43% of the time that the CPU goes to the ready queue for a new strand, the strand at the head of the queue was enabled 2 cycles ago. If there is nothing in the queue, it is counted as a failure (and not shown in the graph). Each successive cycle is counted as a separate failure until something appears in the queue, and each failure is also counted as a load/store stall in Figures 14 and 15. As can be seen from the Figure, an enabled strand has a high probability of waiting several cycles in the queue. In other words, the execution pipeline is generally kept busy processing other strands enabled earlier. The failure rate for queue accesses is only 10%, even given our pessimistic way of counting multiple failures. Furthermore, one can take advantage of the waiting time of a strand to perform preparation of its execution such as prefetching, predecoding, etc. – a topic beyond the scope of this report. It is interesting to see that the performance effectively does not suffer with any branch stalls as indicated earlier. Again, branch delays are hidden by the availability of other active strands being executed in the processing pipeline. (Some branch stalls are masked by superceded load/store stalls.)

Overheads: The overheads due to stranding can be roughly estimated when we fully enable the cache. Since the input data size is chosen such that all memory references are cache hits, the memory latency hiding due to stranding has no advantage over the sequential code. In this worst case scenario, we indeed observed that the stranded version runs modestly slower (15%) than the sequential version due to the additional overhead of the stranded code (e.g., extra instructions for implementing strand synchronization and termination). However, the performance penalty above is inflated since in our simulation each extra superstrand operation is implemented by extending the i860 instruction set (for instance, a separate instruction for terminating a strand). More importantly, in a realistic situation, there cannot be a 100% cache hit, and we anticipate that the performance gain due to latency hiding can easily cover the loss due to the extra overhead.

6 Related Work

In the past, a number of other architectures have been proposed to deal with the increasing demand for higher ILP by increasing on-chip parallelism through multithreading [1, 2, 5, 11, 15, 17, 21, 25].

A natural extension to conventional superscalar architectures is Simultaneous Multithreading [38, 9, 37]. These 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. On the other hand, the Multiscalar architecture [31], Trace Processor [27] and the Multicluster Architecture [10] hide architecture latencies through aggressive control, memory and data speculation.

Compiling for multithreaded architectures is a relatively new field. Some thread partitioning algorithms have been proposed for functional languages [7, 16, 29, 28]. Recently, partitioning algorithms have been developed for imperative languages in which various heuristics were designed based on a cost model for multithreaded architectures [14, 33].

7 Conclusions and Future Work

In this paper, we have proposed the Superstrand program execution model. A unique feature of this model is that a strand, once scheduled for execution, can proceed to completion without stalls due to latencies for branching or data access. This feature simplifies the Superstrand architecture design.

We have presented a compiler framework for partitioning programs into strands. We have also proposed an architecture for efficiently implementing Superstrand synchronization. This architecture makes use of "explicit" dependence information (determined at compile time) in the

Rigged Scoreboard for handling inter-strand synchronization and scheduling. This is in contrast to classical register scoreboarding where implicit dependence information in the sequential code needs to be discovered and resolved at runtime.

In the future, we intend to consolidate the Superstrand program execution model and investigate various architecture features for its efficient implementation. We also intend to complete the implementation of the code generator and related compile optimization techniques. Finally, we plan to conduct a thorough study of the proposed architecture with extensive simulation on a suitable set of benchmark programs and real applications.

Acknowledgments

The authors acknowledge the support from DARPA, NSA, NSF (MIPS-9707125), and NASA. 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, for their valuable input in several discussions.

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] Martin C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines.* PhD thesis, Princeton University Department of Computer Science, June 1996.

[5] Derek Chiou, Boon S. Ang, Robert Greiner, Arvind, James C. Hoe, Michael J. Beckerle, James E. Hicks, and Andy Boughton. StarT-NG: Delivering seamless parallel computing. In Seif Haridi, Khayri Ali, and Peter Magnusson, editors, *Proceedings of the First International EURO-PAR Conference*, number 966 in Lecture Notes in Computer Science, pages 101–116, Stockholm, Sweden, August 29–31, 1995. Springer-Verlag.

[6] Intel Corporation. *i860 Microprocessor Family Programmer's Reference Manual*. Santa Clara, California, 1992. Order number 240875-002.

[7] 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.

[8] 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.

[9] Pradeep K. Dubey, Kevin O'Brien, Kathryn O'Brien, and Charles Barton. Single-program speculative multithreading (SPSM) architecture: Compiler-assisted fine-grained multithreading. In Lubomir Bic, Wim Böhm, Paraskevas Evripidou, and Jean-Luc Gaudiot, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 109–121, Limassol, Cyprus, June 27–29, 1995. ACM Press.

[10] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 149–159, Research Triangle Park, North Carolina, December 1–3, 1997. IEEE-CS TC-MICRO and ACM SIGMICRO.

[11] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine multicomputer. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 146–156, Ann Arbor, Michigan, November 29–December1, 1995. IEEE-CS TC-MICRO and ACM SIGMICRO.

[12] Gesellschaft für Mathematik und Datenverarbeitung mbH. *MANNA Hardware Reference Manual*. Berlin, Germany, 1993.

[13] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, number 757 in Lecture Notes in Computer Science, pages 406–420, New Haven, Connecticut, August 3–5, 1992. Springer-Verlag. Published in 1993.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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

[21] Yuetsu Kodama, Hirohumi Sakane, Mitsuhisa Sato, Hayato Yamana, Shuichi Sakai, and Yoshinori Yamaguchi. The EM-X parallel computer: Architecture and basic performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 14–23, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 23(2), May 1995.

[22] Jack L. Lo, Susan J. Eggers, Henry M. Levy, Sujay S. Parekh, and Dean M. Tullsen. Tuning compiler optimizations for simultaneous multithreading. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 114–124, Research Triangle Park, North Carolina, December 1–3, 1997. IEEE-CS TC-MICRO and ACM SIGMICRO.

[23] 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.

[24] 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.

[25] Kazuaki Okamoto, Shuichi Sakai, Hiroshi Matsuoka, Takashi Yokota, and Hideo Hirono. Multithread execution mechanisms on RICA-1 for massively parallel computation. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 116–121, Boston, Massachusetts, October 20–23, 1996. IEEE Computer Society Press.

[26] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[27] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 138–148, Research Triangle Park, North Carolina, December 1–3, 1997. IEEE-CS TC-MICRO and ACM SIGMICRO.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

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

[37] Jean-Yuan Tsai and Pen-Chung Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 35–46, Boston, Massachusetts, October 20–23, 1996. IEEE Computer Society Press.

[38] 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.