THE EXPANDABLE SPLIT WINDOW PARADIGM FOR EXPLOITING FINE-GRAIN PARALLELISM

Manoj Franklin and Gurindar S. Sohi
Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706

Abstract

We propose a new processing paradigm, called the Expandable Split Window (ESW) paradigm, for exploiting fine-grain parallelism. This paradigm considers a window of instructions (possibly having dependencies) as a single unit, and exploits fine-grain parallelism by overlapping the execution of multiple windows. The basic idea is to connect multiple sequential processors, in a decoupled and decentralized manner, to achieve overall multiple issue. This processing paradigm shares a number of properties of the restricted dataflow machines, but was derived from the sequential von Neumann architecture. We also present an implementation of the Expandable Split Window execution model, and preliminary performance results.

1. INTRODUCTION

The execution of a program, in an abstract form, can be considered to be a dynamic dataflow graph that encapsulates the data dependencies in the program. The nodes of the graph represent *computation* operations, and the arcs of the graph represent *communication* of values between the nodes. The execution time of a program is the time taken to perform the computations and communication in its dynamic dataflow graph. If there are altogether n communication arcs in the graph, and at a time a maximum of m communication arcs can be carried out, then the execution time involves a minimum of n/m communication steps, no matter how many parallel resources are used for computations. With the continued advance in technology, switching components become smaller and more efficient. The effect is that computation becomes faster and faster. Communication speed, on the other hand, seems to be restricted by the speed of light, and eventually becomes the major bottleneck.

The abstract dataflow graph hides information about the distances involved in the inter-node or inter-operation communication. When a dataflow graph is mapped onto a processing structure, interesting dynamics are introduced into the picture. First, depending on the matching between the dataflow graph and the processing structure, adjacent nodes in the graph (those directly connected by a communication arc) may get mapped to either the same processing element, physically adjacent (communication-wise adjacent) processing elements, or distant processing elements. Based on the mapping used, the communication arcs of the graph get "stretched" or "shrunk", causing changes in the communication cost because it takes more time to send values over a distance. Second, since there can only be a finite amount of fast storage for temporarily storing the intermediate computation values, the values have to be either consumed immediately or stored away into some form of backup storage (for example, main memory), creating more nodes and communication arcs.

An inspection of the dynamic dataflow graph of many sequential programs reveals that there exists a large amount of theoretically exploitable instruction-level parallelism [2,3,5,13,24], *i.e.*, a large number of computation nodes that can be executed in parallel, provided a suitable processor model with a suitable communication mechanism backed up by a suitable temporary storage mechanism exists. Where is this parallelism most likely to be found in a dynamic execution of the sequential program? Because most programs are written in an imperative language for a sequential machine with a limited number of architectural registers for storing temporary values, it is quite likely that instructions of close proximity are data dependent. This means that most of the parallelism can be found only further down in the dynamic instruction stream. The obvious way to get to that parallelism is to use a large window of dynamic instructions. This motivates the following bipartite question: "What is the best way to identify a large number of independent operations every cycle, especially if they are to be extracted from a large block of operations intertwined with intricate dependencies, and at the same time reduce the communication costs and the costs of storing temporary results?" The design of a fine-grain parallel processor, to a large extent, revolves around how one attempts to answer this question and the train of questions that arise while attempting to answer it. A good scheme should optimize not only the number of operations executed in parallel, but also the communication costs by reducing the communication distances and the temporary storing away of values, thereby allowing expandability. We consider a processor model to be expandable if its abilities can be expanded easily, as hardware and software technology advances. This requires the processor model to have no centralized resources that can become potential bottlenecks.

Several processing paradigms have been proposed over the years for exploiting fine-grain parallelism. The most general one, the dataflow model, considers the entire dataflow graph to be a single window and uses an unconventional programming paradigm to expose the maximum amount of parallelism present in the application. Such a general model allows the maximum number of computation nodes to be active simultaneously, but incurs a large performance penalty due to the "stretching" of the communication arcs. It also suffers from the inability to express critical sections and imperative operations that are essential for the efficient execution of operating system functions, such as resource management [5,17]. (Of late, more restricted forms of dataflow architecture have been proposed [9,14,16,17].) This is where the power of sequentiality comes in. By ordering the computation nodes in a suitable manner, sequentiality can be used to introduce (and exploit) different kinds of temporal locality to minimize the costs of communication and intermediate result storage. This has the effect of combining several adjacent nodes into one, because the communication arcs between them become very short when temporal locality is exploited by hardware means. A good example of this phenomenon are the vector machines, which exploit the "regular" type of parallelism found in many numeric applications effectively by hardware means such as *chaining* (*i.e.*, forwarding a result directly from a producer to a consumer without intermediate storage).

Sequential execution can be augmented to exploit the "irregular" type of parallelism found in most non-numeric applications. Superscalar processors [10, 12, 15, 18] and VLIW processors [4] do exactly this; they stay within the realm of sequential execution, but attempt to execute multiple operations every cycle. For achieving this, superscalars scan through a window of (sequential) operations every cycle and dynamically detect independent operations to be issued in a cycle. These von Neuman based machines use the conventional programming paradigm, but require a sufficiently large centralized window (obtained by going beyond several basic blocks), if even moderate sustained issue rates are to be desired. The hardware required to extract independent instructions from a large centralized window and to enforce data dependencies typically involves wide associative searches, and is non-trivial. Furthermore, superscalars require multi-ported register files and wide paths from the instruction cache to the issue unit. Although dynamic scheduling with a large centralized window has the potential for high issue rates, a realistic implementation is not likely to be possible because of its complexity, unless novel schemes are developed to reduce the complexity.

VLIW processors partially circumvent this problem by detecting parallelism at compile time, and by using a large instruction word to express the multiple operations to be issued in a cycle. A major limitation of static scheduling is that it has to play safe, by always making worst-case assumptions about information that is not available at compile time. VLIW processors also require some of the centralized resources required by superscalars, such as the multi-ported register files, crossbars for connecting the computation units, and wide paths to the issue unit. These centralized resources can easily become bottlenecks and be a severe limitation on the performance of these machines.

Looking at these existing models, we see that the concept of sequential execution with a large window is good, but definitely not sufficient by itself. The following additional criteria are very important.

- The creation of the large window should be accurate. That is, the window should consist mostly of instructions that are guaranteed to execute, and not instructions that "might" be executed.
- Factors such as the ability to feed instructions into this window, should not be a limitation. That is, it should be possible to get farther down in the instruction stream, without fetching and decoding the instructions in between.
- There should be provision to issue loads before proper address disambiguation, *i.e.*, in a speculative manner.

We also need a powerful way of decentralizing the critical resources in the system. Our objective is to use the conventional sequential programming paradigm with dynamic scheduling and large windows, but augmented with novel techniques to decentralize the critical resources. This is achieved by splitting a large operation window into small windows, and executing many such small windows in parallel. The principle of dataflow is used in a restricted manner to pass values efficiently across the multiple windows in execution; the execution model within each window can be a simple, sequential processor. As we will see later in this paper, such an approach has the synergistic effect of combining the advantages of the sequential and the dataflow execution models, and the advantages of static and dynamic scheduling.

1.1. Organization of the Paper

We have outlined the important issues pertaining to designing a high-performance fine-grain parallel processor. The rest of this paper is organized as follows. Section 2 describes the basic philosophy behind the Expandable Split Window (ESW) processing paradigm. Section 3 describes a possible implementation of this paradigm. The description includes details of the instruction issue mechanism (distributed instruction caches), the distributed register file, the distributed memory disambiguation unit, and the distributed data cache. Section 4 presents preliminary performance results for the new paradigm, obtained from a simulation study using the SPEC benchmarks. Section 5 provides a summary of the research done so far, and the future course of work.

2. THE EXPANDABLE SPLIT WINDOW PROCESSING PARADIGM

To have an expandable system that can handle large windows in an efficient manner, and whose ability can be expanded easily as technology advances, we need to decentralize all the critical resources in the system. These include the hardware for dependency enforcement and identification of independent operations within the window, the instruction supply mechanism, and the memory address disambiguation mechanism. A convenient way of simplifying the hardware for identification of independent operations is to split the large window into smaller windows (c.f. Figure 1) so that the task of searching a large window can be split into 2 smaller subtasks: (i) independent searches (if need be) in small windows, all of which can be done in parallel, and (ii) enforcement of control and data dependencies between the smaller windows. The dynamic scheduling hardware can then be divided into two hierarchical units — a distributed top-level unit that enforces dependencies between the small windows, and

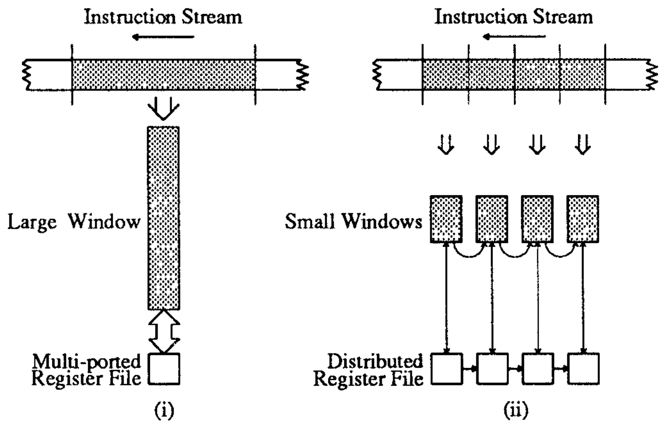

Figure 1: Splitting a large window of instructions into smaller windows
(i) A single large window (ii) A number of small windows

several independent lower-level units at the bottom level, each of which enforces dependencies within a small window and identifies the independent instructions in that window. Each of these lower-level units can be a separate execution unit akin to a simple (possibly sequential) execution datapath.

Having decided to split the large operation window, the next question is where to split. There are several issues to be considered here. First, the overall large window would invariably consist of many basic blocks, obtained through dynamic branch prediction. When a prediction is found to be incorrect, we would like not to discard the part of the window before the mispredicted branch. Second, it would be ideal if (at least some of) the information available at compile time is conveyed to the dynamic scheduling hardware. With these views in mind, we propose to consider a single-entry loop-free call-free block of (dependent) instructions (or a **basic window**) as a single unit, *i.e.*, each such block forms a small window[1]. Splitting the *dynamic* instruction stream at such *statically* determined boundaries has the following advantages:

(1) If a basic window is a basic block or a subset of a basic block, then it is a straight-line piece of code that is entirely executed once it is entered, and the registers read and written are the same for any execution of that basic window. Ironically, the dynamic scheduling hardware expends quite a bit of effort to reconstruct this information. This information can be statically determined and conveyed to the hardware in a concise manner (explained in Section 3.3.3), which simplifies the hardware for the enforcement of register value dependencies between the basic windows. If a basic window is a superset of several basic blocks, then also it is possible to express this information, but the information will be conservative, as it has to consider all possible paths through the basic window.

(2) The control flow graph of most programs are embedded with diamond-shaped structures (reconvergent fanouts, typically due to **if-then** and **if-then-else** statements), which tend to have low branch prediction accuracies. In such situations, there is merit in encompassing the diamond-shaped part within a basic window so as to allow subsequent windows to start execution earlier with no fear of being discarded later due to an incorrect branch prediction at an earlier stage.

(3) By merging small basic blocks into one basic window, we can tide over the problem of poor utilization of an execution unit when small, data-dependent basic blocks appear next to each other in the dynamic instruction stream. This situation arises frequently in non-numeric benchmarks, where the mean basic block size is typically 5-6 instructions.

Example: The basic idea behind the new paradigm is best illustrated by an example. Consider the following loop, which adds the number 10 to 100 elements of an array and sets an element to 1000 if it is greater than 1000. This is a typical **for** loop with an **if** statement enclosed within.

[1] If a single-entry loop-free call-free block is too large, or if it is a collection of blocks with few data dependencies between them, then it can be split into appropriate smaller blocks. In this paper, quite frequently we use a basic block as a basic window for illustration purposes, although a basic window could be a subset or a superset of basic blocks. We are still investigating the issues involved in creating a basic window that is larger than a basic block. We are also investigating the implications of removing the "loop-free" requirement. The basic window discussed in this paper should therefore be considered only as one example of a basic window of instructions.

```
A:  R1  =  R1 + 1
    R2  =  [R1, base]
    R3  =  R2 + 10
    BLT  R3, 1000, B
    R3  =  1000
B:  [R1, base]  =  R3
    BLT  R1, 100, A
```

In this example, all the instructions in one iteration except the last instruction can be executed only in strict sequential order. We can consider each iteration of this loop to be a basic window; at run time, the loop gets expanded into multiple basic windows as shown below.

	Basic Window 1		Basic Window 2
A_1:	$R1_1$ = $R1_0$ + 1	A_2:	$R1_2$ = $R1_1$ + 1
	$R2_1$ = [$R1_1$, base]		$R2_2$ = [$R1_2$, base]
	$R3_1$ = $R2_1$ + 10		$R3_2$ = $R2_2$ + 10
	BLT $R3_1$, 1000, B_1		BLT $R3_2$, 1000, B_2
	$R3_1$ = 1000		$R3_2$ = 1000
B_1:	[$R1_1$, base] = $R3_1$	B_2:	[$R1_2$, base] = $R3_2$
	BLT $R1_1$, 100, A_2		BLT $R1_2$, 100, A_3

Multiple instances of a register are shown with different subscripts, for example, $R1_1$ and $R1_2$. Although the instructions of a basic window in this example are sequentially dependent, a new basic window can start execution once the first instruction of the previous basic window has been executed. Our idea is to execute these multiple basic windows in parallel, with distinct execution units. Notice that among the two branches in an iteration, branch prediction is performed only for the second one, which can be predicted much more accurately than the other. The low confidence branch (the first one) has been incorporated within the basic window so that a poor branch prediction does not result in an inaccurate dynamic window. For a general program, the compiler divides the instruction stream into basic windows based on the following factors: (i) the pattern of data dependencies between instructions, (ii) maximum size allowed for a basic window, and (iii) predictability of branch outcomes.

3. IMPLEMENTATION OF THE ESW PARADIGM

As mentioned earlier, to be expandable, there should be no centralized resources that could become potential bottlenecks. Designing a dynamically scheduled fine-grain parallel processor with decentralized resources poses several challenges, some of which are: (i) an adequate decentralized instruction issue mechanism, (ii) adequate CPU resources, and (iii) an adequate decentralized memory system. Other issues, of no less magnitude, that pose challenges especially when doing speculative execution are: (i) efficient means of forwarding results from one instruction to another when a number of instructions are simultaneously active, (or efficient means of detecting and enforcing register dependencies), (ii) disambiguating memory addresses and enforcing memory dependencies, (iii) good branch handling/prediction schemes that allow the dynamic window to be expanded fairly accurately, and (iv) efficient mechanisms to recover a precise state when special events such as incorrect branch predictions and exceptions occur.

In this section, we discuss a possible implementation of the new processing paradigm. In the ensuing subsections, it will

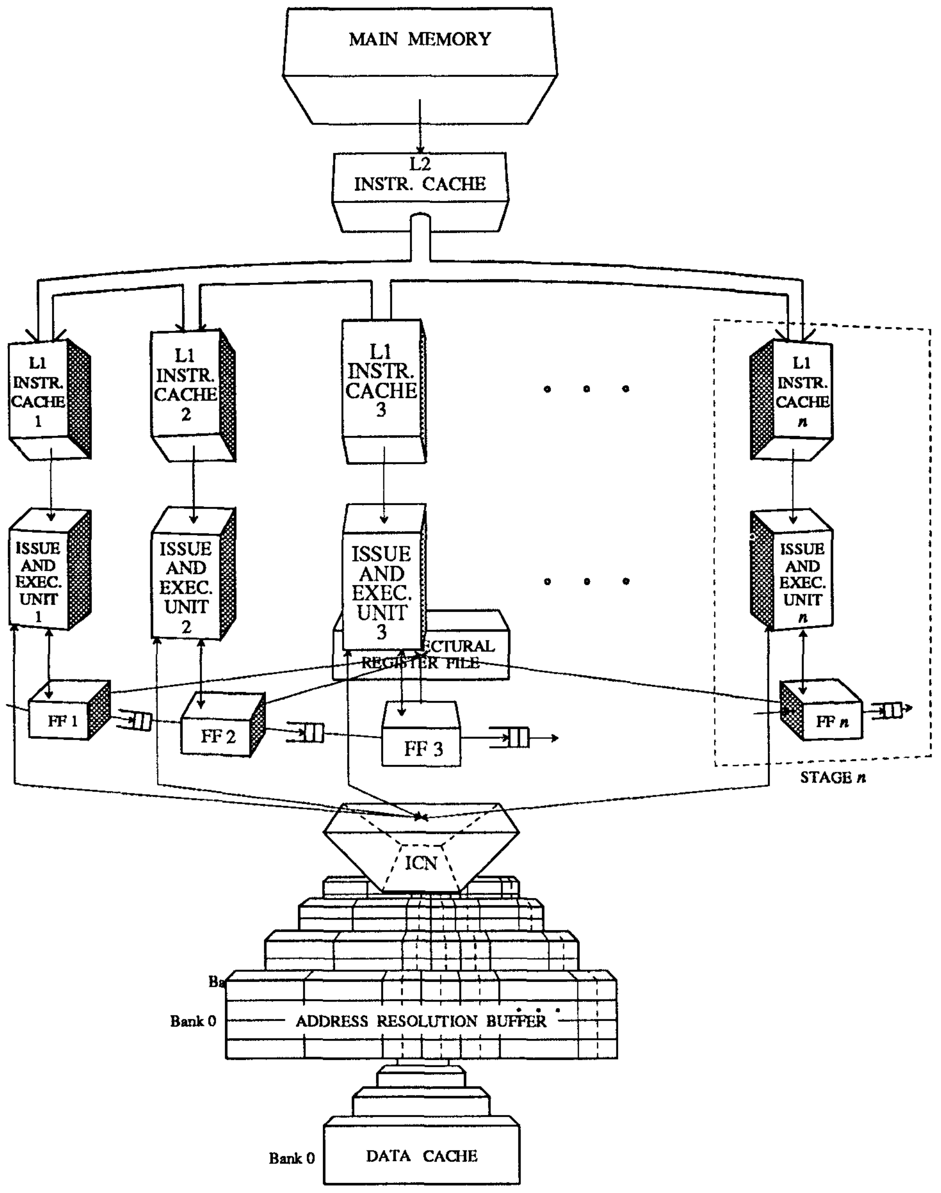

Figure 2: Block Diagram of the Expandable Split Window Paradigm Implementation

become evident that throughout the design we have emphasized two points — *decentralization* (which facilitates expandability) and *realizability*. Several novel techniques have been used to decentralize the resources, without which the potential of the new paradigm could not have been exploited. The techniques used for decentralizing the different parts of the system are different, because their workings and the way they fit into the system are different. As the purpose of this paper is to introduce the paradigm and present an overall view, and because of space restrictions, some of the intricate design details are not presented here.

Figure 2 presents the block diagram of our machine. The processor consists of several independent, identical *stages*, each of which is equivalent to a typical datapath found in modern processors. The stages conceptually form a circular queue, with hardware pointers to the head and tail of the queue. These pointers are managed by a control unit (not shown in Figure 2 for clarity), which also performs the task of assigning basic windows to the stages. Dynamic branch prediction is used (if required) to decide new basic windows, and every cycle, the control unit assigns a new window to a stage unless the circular stage queue is full. It is important to note that all that the control unit does when it assigns a window to a stage unit is to tell the stage to execute a basic window starting at a particular PC (program counter) value; it is up to the stage to fetch the required instructions, decode them and execute them (most likely in serial order) until the end of the basic window is reached. The control unit does not perform instruction decoding. (A major purpose of "decoding" instructions in a superscalar processor is to establish register dependencies between instructions. We shall see in section 3.3 how we enforce the register dependencies without dynamically decoding the instructions.) Because the task of the control unit is relatively straightforward, it does not become a potential bottleneck. (Control units with instruction decoders that feed centralized windows are a major impediment to performance in superscalar processors, as shown in [21].)

The active stages, the ones from the head to the tail, together constitute the large dynamic window of operations, and the stages contain basic windows, in the sequential order in which the windows appear in the dynamic instruction stream. When all the instructions in the stage at the head have completed execution, the window is committed, and the control unit moves the head pointer forward to the next stage. Consequently, the big window is a *sliding* or *continuous* window, and not a fixed big window, a feature that allows more parallelism to be exploited [24]. The major parts of the ESW implementation are described below.

3.1. Distributed Issue and Execution Units

Each stage has as its heart an Issue and Execution (IE) unit, which takes operations from a local instruction cache and pumps them to its functional units after resolving their data dependencies. An IE unit is comparable to the Instruction Issue and Execution Unit of a conventional processor in that it has its own set of functional units. Notice that if the cost of functional units (especially the floating point units) is a concern, infrequently used functional units may be shared by multiple stages. It is also possible to have a small interconnect from the IE units to a common Functional Unit Complex. In any given cycle, up to a fixed number of ready-to-execute instructions begins execution in each of the active IE units. It is possible to have out-of-order execution in an IE unit, if desired.

3.2. Distributed Instruction Supply Mechanism

The proposed scheme for exploiting instruction-level parallelism by the execution of multiple basic windows in parallel will bear fruit only if instructions are supplied to the IE units at an adequate rate. Supplying multiple IE units in parallel with instructions from different basic windows can be a difficult task for an ordinary centralized instruction cache. We need novel instruction cache designs, ones that are suited to the issue strategy and the execution model used.

We propose to use a two-level instruction cache, with the level 1 (L1) (the level closest to the IE units) split into as many parts as the number of stages, one for each stage, as shown in Figure 2. An IE unit accesses instructions from its L1 cache. If a request misses in the L1 cache, it is forwarded to the L2 instruction cache. If the window is available in the L2 cache, it is supplied to the requesting L1 cache (a fixed number of instructions are transferred per cycle until the entire window is transferred — a form of intelligent instruction prefetch). If the transferred window is a loop, the L1 caches of the subsequent stages can also grab the window in parallel, much like the *snarfing* (*read broadcast*) scheme proposed for multiprocessor caches [8]. If the request misses in the L2 cache, it is forwarded to main memory. Notice that several IE units can simultaneously be fetching instructions from their corresponding L1 caches, and several L1 caches can simultaneously be receiving a basic window (if the window is a loop) from the L2 cache, in any given cycle.

3.3. Distributed Inter-Instruction Communication Mechanism

A high-speed inter-instruction communication mechanism is central to the design of any processor. Earlier we saw that a centralized register file can become a bottleneck for the VLIW and superscalar processors. In our processor, at any one time there could be more than 100 active operations, many of which may be executed simultaneously. Clearly, a centralized architectural register file cannot handle the amount of register traffic needed to support that many active operations; we need a decentralized inter-instruction communication mechanism. The main criteria to be considered in coming up with a good decentralized scheme is that it should tie well with the distributed, speculative execution feature of our model. (For instance, the split register file proposed for the VLIW and superscalars cannot be of much help to our model.)

In order to do a careful design of the decentralized register file, we first conducted a quantitative study of the traffic handled by the architectural registers in a load/store architecture (MIPS R2000). In particular, we studied how soon newly created register instances are used up by subsequent instructions and are eventually overwritten. These studies showed that: (i) a significant number of register instances are used up in the same basic block in which they are created, and (ii) most of the rest are used up in the subsequent basic block. The first result implies that if we have a local register file for each stage, much of the register traffic occurring in a stage can be handled by the local register file itself. Only the last updates to the registers in a stage need be passed on to the subsequent stages. The second result implies that most of these last updates need not be propagated beyond one stage. Thus we can exploit the *temporal locality of usage of register values* to design a good decentralized register file structure that ties well with our execution model, and that is exactly what we do.

3.3.1. Distributed Future File

In our proposed design, each stage has a separate register file called a *future file*. These distributed future files are the working files used by the functional units in the IE units. In that sense, they work similar in spirit to the *future file* discussed in [20] for implementing precise interrupts in pipelined processors. As we will see in section 3.5, the distributed future file simplifies the task of recovery when incorrect branch predictions are encountered. If out-of-order execution within IE units is desired, then each IE unit should also have some means of forwarding results within the unit (possibly reservation stations [23]). Another advantage of the distributed future file structure in that case is that it allows independent register renaming for each stage. An architectural register file is maintained to facilitate restart in the event an exception occurs in the IE unit at the head of the queue, and to bring the processor to a quiescent state before system calls.

3.3.2. Register Data Dependencies Within a Stage

Register dependencies within a basic window are enforced by either doing serial execution within an IE unit, or using reservation stations (or renamed registers) with data forwarding.

3.3.3. Register Data Dependencies Across Stages

The burden of enforcing register data dependencies across multiple basic windows becomes light when windows are ordered in a sequence, and the data flowing into and out of basic window boundaries are monitored. To keep the discussion simple, we shall explain the working of the scheme for the case when basic windows are either basic blocks or subsets of basic blocks. For a given basic block, the registers through which externally-created values flow into the basic block and the registers through which internally-created values flow out of the basic block are invariants for any execution of that basic block. We express these invariants concisely by bit maps called **`use`** and **`create masks`**.

The **create** and **use masks** capture a good deal of the information (related to register traffic) in a basic block in a simple and powerful way. If there were no **create masks**, each instruction in the large dynamic window has to be decoded before identifying the destination register and setting the corresponding "busy bit" in the register file. Subsequent instructions in the overall large window, even if independent, have to wait until all previous instructions are decoded, and the "busy bits" of the appropriate registers are set. The advantage of having a **create mask** is that all the registers that are written in a basic block are known immediately after the mask is fetched, *i.e.*, even before the entire basic block is fetched from the instruction cache and decoded. (As mentioned earlier, this "decoding" problem is a major problem in the superscalar processors proposed to date.) Independent instructions from subsequent basic windows can thus start execution, possibly from the next cycle onwards, and the hardware that allows that is much simpler than the hardware required to decode a large number of instructions in parallel and compare their source and destination registers for possible conflicts.

Generation of Register Masks

If adequate compile-time support is available, the masks can be generated by the compiler itself[2]. If the use of compile-time support is not an option (for example if object code compatibility is required), or if the additional code space overhead due to the masks is considered intolerable, the masks can be generated at run time by hardware the first time the block is encountered, and stored in a dynamic table for later reuse.

Forwarding of Register Values

When an instruction completes execution, its results are forwarded to subsequent instructions of that stage. If the result is a register's *last update* in that basic block, the result is written to the future file of that stage, and forwarded to the future files of the subsequent stages as well, one stage at a time. When a result from a previous stage reaches a future file, the appropriate register is updated. If the register entry appears in the **use mask** of that stage, then the reservation stations in that unit are checked for possible matchings. The result is also forwarded to the subsequent future file in the next cycle. The **create** and **use masks** helps to reduce the forwarding traffic and the associative search involved in forwarding the results. When the result of an instruction (*i.e.*, a new register instance) is forwarded to a future file, the reservation stations in the corresponding IE unit need be searched for possible matchings only if the register entry appears in its **use mask**. Similarly, if the register entry appears in a **create mask**, then the result need not be forwarded to the subsequent stages, because they need a different instance of that register[3]. Notice that data from several stages can simultaneously be traveling to subsequent stages in a pipelined fashion. Queues are used to facilitate this forwarding.

[2] All optimizing compilers invariably do dataflow analysis [1,7]; the `create` and `use` masks are similar to the `def` and `use` variables computed by these compilers for each basic block, except that the former pair represent architectural registers and the latter pair represent variables of the source program.

[3] Since most of the register instances are used up either in the same basic block in which they are created or in the subsequent basic block, we can expect a significant reduction in the forwarding traffic because of the `create mask`.

When the IE unit at the head commits, the last updates in that window are written to the architectural register file for purposes of precise state recovery (see section 3.5).

3.4. Distributed Data Memory System

When a processor attempts to issue and execute many instructions in parallel, it is imperative that the memory system should be capable of supporting multiple memory references per cycle, preferably with small latencies [22]. The latency can be reduced by using a data cache. In line with our objective of expandability, the memory system also has to be decentralized.

Decentralizing the memory system is harder than most of the other parts. When a system performs speculative execution, a store operation can be allowed to proceed to the memory system only when it is guaranteed to commit; otherwise the old memory value is lost and recovery will not be possible. Nevertheless, succeeding loads (from speculatively executed code) to the same location do require the new value, and not the old value. Thus, there must be some means of forwarding "uncommitted" memory values to subsequent loads, just like the forwarding of "uncommitted" register values. In the case of register values, the distributed register files served as good temporary platforms to hold these "uncommitted" register values, and the **create** and **use masks** served to enforce the dependencies between them. But such a straightforward replication scheme will not work for the memory system, because of the following reasons.

(1) Even if replication of data caches was feasible, we still have the problem of maintaining consistency among the multiple copies, and that too, with the restriction that values have to be written onto a memory location in the order given by the sequential semantics of the program.

(2) With register values, maintaining this consistency was easily done by using the static **create** and **use masks**. In the case of memory values, the number of memory locations is much too large to permit the use of such bitmaps. Furthermore, memory addresses, unlike register addresses, are computed at run time, and therefore have the *aliasing* problem. Although some addresses could be disambiguated statically, many others, especially those arising from pointer variables in the source program, can be determined only at run time. This is reflected in the fact that even if the data cache is decentralized, we still need to do a global memory disambiguation within the large active window.

3.4.1. Enforcing Memory Data Dependencies

To guarantee correctness of execution, before a load operation is issued, the load address has to be checked (for possible conflicts) with the addresses of all pending stores in the same basic window as well as the preceding active basic windows. Whereas performing this (centralized) associative search is one problem, the more serious problem is that the memory addresses of some of the previous stores may not be determined yet. If worst-case assumptions are made about possible memory hazards and a load operation is made to wait until the addresses of all pending stores are determined, parallelism is inhibited and performance might be affected badly. What we need to do is to allow for the execution of load instructions without proper disambiguation, with facilities provided for recovery actions when it is determined that an incorrect value has been loaded. For this recovery, we can use the same facility that has been provided for fast recovery in times of incorrect branch prediction. Our analysis of the memory traffic in programs indicates that most of the stored values are not reloaded in the immediate future, say within the next 100 instructions (a good compiler would keep soon-to-be-used values in registers). Therefore, we expect that most of the time, the unresolved loads will fetch the correct values from the data cache/main memory.

To detect if an unresolved load fetched an incorrect value, we need to perform global memory disambiguation within the large active window. On first glance, this appears to be a sequential process, or one requiring a large associative search; but we have developed a decentralized scheme for carrying this out. The scheme, called an *Address Resolution Buffer (ARB)*, is the solution that we settled on after considering several other options.

3.4.2. Interleaved Address Resolution Buffer

As with all other major resources, the ARB is also decentralized. The ARB is a special cache for storing information relevant to the loads and stores that have been issued from the active large window. It is interleaved (based on the memory address) to permit multiple accesses per cycle. Figure 3 shows the block diagram of a 2-way interleaved ARB. Each ARB bank would typically have provision for storing 4 - 8 addresses and related information. Associated with each address entry is a bitmap with twice as many bits as the number of stages. Of the 2 bits per stage, one is used for indicating if a (partially resolved) load has been performed from the corresponding stage, and the other is for indicating if a store has been performed from that stage. There is also a *value* field associated with each address entry to store one memory value for that location.

Within each stage, memory disambiguation is done in a strict serial order with the help of a local queue-like structure, which also has provision for forwarding values from stores to loads within that stage. When a load is properly resolved locally within its stage (automatic if each unit executes sequentially), it is ready to be issued. When the load is issued, the addresses of the entries in the appropriate ARB bank are associatively checked (notice that this associative search is only 4-way or 8-way, searching for a single key) to see if the same address is present in the ARB. If the address is not present, an ARB entry is allotted to the new address. If the address is present in the ARB, then a check is made in its bitmap to see if an earlier store has been made to the same address from a previous stage. If so, then it takes the value stored in the *value* field of the entry. Finally, the load bit corresponding to the stage is set to 1.

When a store is performed, a similar associative check is performed to see if the address is already present in the ARB. If not, an entry is allotted to the new address. The bitmap corresponding to the store address is updated to reflect the fact that a store has been performed from a particular stage. The value to be stored is deposited in the *value* field of the bitmap. If the store address was already present in the ARB, then a check is also performed to see if an earlier load has been made to the same address from a succeeding stage. If so, recovery action is initiated so that all stages beyond the one that made the incorrect load are nullified, and restarted. Since there is provision for storing only one "store value" at a time for any address, special recovery actions are taken when a second store request occurs to an address.

When the stage at the head is retired, all load and store marks corresponding to that stage are erased immediately. This requires a facility to clear all the bits in 2 columns of the ARB in 1 cycle. Similarly, when the tail pointer is moved backwards during recovery, the columns corresponding to the stages stepped over by the tail pointer are also cleared immediately. An ARB entry is reclaimed for reuse when all the load and store bits associated with the entry are cleared.

The ARB scheme has the full power of a hypothetical scheme that performs associative compares of the memory addresses in the entire large window. Here also, we use the concept of splitting a large task (memory disambiguation within a large window) into smaller subtasks. Furthermore, it ties well with our execution model. It allows speculative loads and speculative stores! It also allows forwarding of memory values when loads are performed, and all stores are effectively "cache hits" because the value is stored only in the ARB until its stage is committed.

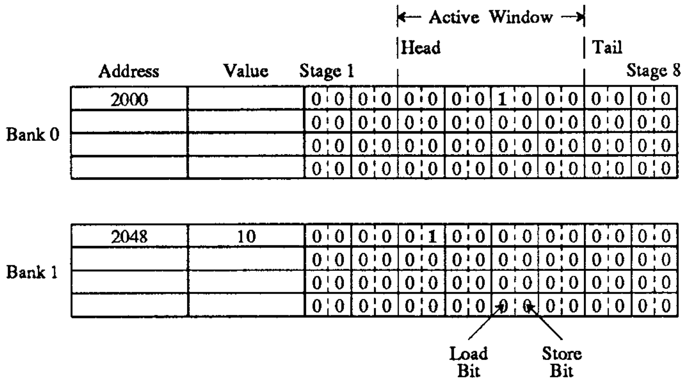

Figure 3: A Two-Way Interleaved Address Resolution Buffer

We feel that the ARB is a very powerful, decentralized (and thus expandable) way of dynamically disambiguating memory references. We are considering the performance impact of set-associative mappings in the ARB, as well as other performance implications of this "cache besides the cache".

3.4.3. Interleaved Data Cache

As seen from the discussion so far, conventional data caches are not appropriate for the ESW paradigm; the exact nature of the data caches is a subject of our future research. Currently we propose to use an interleaved non-blocking cache similar to the one proposed in [22].

3.5. Enforcing Control Dependencies

Dynamic branch prediction is used (if required) to fetch new basic windows. When a conditional branch instruction or a return instruction is executed in an IE unit, its outcome is compared with the earlier prediction. If there is a mismatch, then all subsequent windows are discarded. This is easily done by advancing the tail pointer (of the circular stage queue) to the stage that succeeds the one containing the mispredicted branch. In the next cycle, the IE unit at the tail starts fetching the correct set of instructions. Notice that the distributed future file system helps to maintain precise states at each basic window boundary, which eases the implementation of the recovery mechanism significantly.

4. PERFORMANCE ISSUES

4.1. Preliminary Experimental Results

We are in the process of conducting several simulation studies, both to verify the potential of the new processor design and to study the effects of compiler optimizations. A sound evaluation of the execution model can be done only after the development of a compiler that considers the idiosyncrasies of the model and exploits its potential. In this section, we present some of the preliminary results obtained in our simulation studies. These results should be viewed as a realistic starting point.

The simulator that we developed uses the MIPS R2000 - R2010 instruction set and functional unit latencies. All important features of the ESW paradigm, such as the split instruction caches, distributed address resolution buffer, and data caches, have been included in the simulator. The simulator accepts executable images of programs, and executes them; it is not trace driven. Its features are listed below:

- Number of stages can be varied.
- Up to 2 instructions are fetched/decoded/issued from each of the active IE units, every cycle; out-of-order execution is used in each stage.
- A basic window can have up to 32 instructions.
- The data cache is 64Kbytes, direct-mapped, and has an access latency of **2 cycles** (one cycle to pass through the interconnect between the IE units and the cache, and another to access the ARB and the cache in parallel). The interleaving factor of the data cache and the ARB is the smallest power of 2 that is equal to or greater than twice the number of stages. (For instance, if the number of stages is 6, the interleaving factor used is 16.) The cache miss latency is 4 cycles.
- Each stage has a 4Kword L1 instruction cache.
- The L2 instruction cache has not been included in the simulator; instead, we assume 100% hit ratio for the L2 instruction cache.
- The branch prediction mechanism for conditional branches uses the 3-bit counter scheme proposed in [19]. For effectively predicting the return addresses of procedure calls, there is a stack-like mechanism similar to the one discussed in [11], with a stack depth of 20.

The simulator is not equipped to detect basic windows that are supersets of basic blocks (which needs a control flow analysis within each procedure to make sure that each basic window has only a single entry); currently it uses basic blocks (or smaller blocks if a basic block is larger than 32) as basic windows.

For benchmarks, we used the SPEC benchmark suite. The benchmark programs were compiled for a DECstation 3100; notice that the `a.out` executables so obtained have been compiled for a single-IPC machine. The C benchmarks were simulated to completion; the FORTRAN benchmarks were simulated up to approximately **1 billion** instructions.

Table 1 presents the results obtained with code generated for an ordinary single IPC machine, and with basic blocks used as basic windows. The latter feature affects the performance of the integer benchmarks because they contain several small basic blocks with poor predictability, as is evident from the "Mean Basic Block Size" column and the "Branch Prediction Accuracy" column. The C benchmarks are executed with 4 stages and the FORTRAN benchmarks with 10 stages.

From Table 1 we see that, even using basic blocks as basic windows, we get fairly impressive completion rates, ranging from 1.81 to 2.04 for the C programs, and 1.92 to 5.88 for the FORTRAN benchmarks, even using code that is not compiled/scheduled for our machine model. (Notice that the actual speedup numbers would be higher because the instruction completion rate of a conventional processor is ≤ 1.)

Table 1: Instruction Completion Rates with Unmodified Code

Benchmarks	Mean Basic Block Size	No. of Stages	Branch Prediction Accuracy	Completion Rate
eqntott	4.19	4	90.14%	2.04
espresso	6.47	4	83.13%	2.06
gcc	5.64	4	85.11%	1.81
xlisp	5.04	4	80.21%	1.91
dnasa7	26.60	10	99.13%	2.73
doduc	12.22	10	86.90%	1.92
fpppp	113.42	10	88.86%	3.87
matrix300	21.49	10	99.35%	5.88
spice2g6	6.14	10	86.95%	3.23
tomcatv	45.98	10	99.28%	3.64

In comparing our results to other results in the literature, we see that we are achieving issue rates similar to Butler, *et. al.* [3], with similar resources but larger window sizes (our results are slightly better in several cases), and much better than the issue rates achieved by Smith, *et. al* [21]. This is despite the fact that we include all stalls, specifically stalls due to: instruction and data cache misses (instruction cache miss stalls cause a noticeable degradation in performance in many schemes), data cache bank contention (including contention due to speculative loads), recovery due to incorrect speculative loads (we do not assume disambiguated memory operations), and recovery due to branch prediction with a real predictor (our predictor does not do too well, as can be seen in Table 1), whereas the other studies make optimistic assumptions in these cases.

We tried using more stages for the C benchmarks, and our results improved somewhat, but not significantly (about 20-30% increase in performance) because of our using a basic block as a basic window, and the small average size of a basic block in the code compiled for a single-IPC machine.

4.2. Role of the Compiler

The compiler has the opportunity to play an important role in bringing to reality the full potential of this processing paradigm. The compiler can introduce helpful transformations that are tailored to the idiosyncrasies of the execution model. For instance, if the compiler partitions the instruction stream into basic windows that are larger than basic blocks, or if the compiler knows how the instruction stream would be dynamically partitioned into windows, it can attempt to move up within a window the operations whose results are needed early on in the following window. Similarly, it can perform a static memory disambiguation, and push down (within a window) those loads which are guaranteed to conflict with stores in the previous windows. Many other "optimizations" are also possible. The important ones include:

- Partition the program into basic windows that are supersets of basic blocks and convey this to the hardware. This would allow a more accurate expansion of the dynamic window to get at more parallelism. (This information is present in the control flow graph constructed by the compiler.) In case the basic block is large, decide where to split.

- Perform static code re-scheduling based on the idiosyncrasies of the ESW model, as described earlier. For example, pack all *dependent* instructions into a basic window.

Table 2 gives a feel of the improvements we can expect to see when some of the above points are incorporated. The results in Table 2 were obtained by manually performing the optimizations in one or two important routines in some of the benchmarks. Hand code scheduling was performed only for those benchmarks which spend at least 50% of the time in a limited portion of the code, so that manual analysis is possible. The only transformations that were performed are: (i) moving up within a basic window those instructions whose results are needed in the critical path in the following basic windows. (Such instructions typically include induction variables, which are usually incremented at the end of a loop when code is generated by an ordinary compiler.) (ii) in the *cmppt* routine of **eqntott**, we expand a basic window to a superset of 3 basic blocks. (iii) in the *compl_lift* and *elim_lowering* routines of **espresso**, we consider basic windows that are supersets of basic blocks.

Table 2: Instruction Completion Rates with Modified Code

Benchmarks	No. of Stages	Prediction Accuracy	Completion Rate
eqntott	4	95.58%	4.23
	8	96.14%	4.97
espresso	4	92.17%	2.30
dnasa7	10	98.95%	7.17
matrix300	10	99.34%	7.02
tomcatv	10	99.31%	4.49

Even the simple optimizations implemented for Table 2 boost performance considerably in the cases considered — **eqntott** is achieving close to 5 instructions per cycle, and **matrix300** and **dnasa7** are able to sustain over 7 instructions per cycle (the completion rate for **dnasa7** has improved from 2.73 to 7.17 with this simple code scheduling). We expect that with such simple compiler enhancements (for example facilitating the hardware's task of recovering part of the control structure of the program), we will be able to double the performance of the C benchmarks. We base our expectation on an analysis of the C programs which suggests that expanding the basic window into a sequence of basic blocks will not only increase the average size of the basic window (in terms of useful instructions executed) by a factor of about 2, but also allow us to get past of many of the "badly predicted" branches since they become part of the basic window and do not prevent us from accurately expanding the dynamic window. (Notice that the branch prediction accuracies for **eqntott** in Table 2 are much better than those in Table 1 for precisely this reason; the prediction accuracies are different for 4 and 8 stages because the exact dynamic instruction stream that enters the IE units differs with different number of stages.) For the FORTRAN programs, the primary impediment to greater performance in most cases is the hardware limitation, though in some cases, such as **tomcatv**, static memory disambiguation would assist in reducing the cycles lost due to incorrect speculative loads.

5. SUMMARY AND FUTURE WORK

5.1. Summary

We have proposed a new processing paradigm for exploiting fine-grain parallelism. The model, which we call the Expandable Split Window (ESW) paradigm, shares a number of properties with the restricted dataflow machines, but was derived from the von Neumann architecture. The essence of dynamic dataflow execution is captured by simple data forwarding schemes for both register and memory values. The fundamental properties of the von Neumann architecture that we retained includes a sequential instruction stream, which relies on inter-instruction communication through a set of registers and memory locations. The result is a simple architecture that accepts ordinary sequential code, but behaves as a fairly restricted dataflow machine.

We proposed an implementation of the proposed model. In our view, the beauty of the ESW model and its implementation lies in its realizability, not to mention its novelty. It draws heavily on the recent developments in microprocessor technology, yet goes far beyond the centralized window-based superscalar processors in exploiting "irregular" fine-grain parallelism. It has no centralized resource bottlenecks that we are aware of. This is very important, because many existing execution models are plagued by the need for centralized resources. Almost all the parts of our implementation are found in conventional serial processors, the only exception is the Address Resolution Buffer (ARB); yet these parts have been arranged in such a way as to extract much more parallelism than was thought to be realistically possible before. The ARB scheme presented in this paper is, to the best of our knowledge, the first decentralized design for memory disambiguation with a large window of instructions[4].

Another feature of the ESW paradigm is its expandability. When advances in technology allow more transistors to be put on a chip, our implementation can be easily expanded by adding more stages; there is no need to redesign the implementation/architecture.

The performance results are also very promising. We are able to achieve about 2 instructions per cycle, with 4 stages, for the C programs, and more for the FORTRAN programs, with no special code transformations, and taking all stalls that degrade performance into account. When we applied some optimizations suited to the ESW model to some of the benchmarks, the performance was enhanced substantially; the completion rate increased by a factor of 2.5 for **eqntott** and a factor of 2.6 for **dnasa7**. We strongly feel that the ESW model has significant potential for becoming the model of choice for future fine-grain parallel processors. It encompasses many of the strengths of the VLIW, superscalar, decoupled, systolic, and restricted dataflow models of computation, and overcomes most of their drawbacks.

5.2. Future Work

We would like to conduct a series of studies to determine what a basic window of instructions should be. Currently, we use a single-entry call-free loop-free block of instructions as a basic window. We are investigating the implications of removing the "loop-free" requirement. Studies are also needed to determine the optimum number of stages and the maximum number of

[4] There is an ongoing independent work on hardware support for dynamic disambiguation of memory references in the context of a VLIW processor in industry [6].

instructions to be issued from each IE unit in a cycle. Another issue worth investigating is the benefit of out-of-order execution within an IE unit when it contains a straight-line piece of code. If a window consists mostly of dependent operations, serial execution within each IE unit might suffice, thereby saving the associative hardware and reservation stations required within each IE unit to perform out-of-order execution.

The design of our fine-grain parallel processor is at a stage where many of the hardware issues have been investigated, and the arena slowly shifts to software issues. We expect to conduct further analysis of programs for the purpose of software development, especially a suitable compiler for the machine. We feel that the compiler effort would not be as detailed as that required for some other fine-grain parallel models since most of the tasks that we ask of the compiler, such as the construction of the basic window and simple code scheduling are routinely done by compliers, and some of the harder tasks, such as static memory disambiguation are helpful, but not essential to our execution model.

ACKNOWLEDGEMENTS

The foundation for this work was laid in discussions with Jim Smith in the 1988-1989 time frame, and continued discussions with him have helped to refine the work. We would like to thank Mitch Alsup and Kemal Ebcioglu for their insightful comments on the paper, and Joel Emer and Bob Rau for discussions that helped to refine some of the ideas presented in this paper. Financial support for this work was provided by NSF grant CCR-8919635 and by an IBM Graduate Fellowship.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[2] T. M. Austin and G. S. Sohi, "Dynamic Dependency Analysis of Ordinary Programs," *Proc. 19th Annual International Symposium on Computer Architecture*, 1992.

[3] M. Butler, T. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single Instruction Stream Parallelism Is Greater than Two," *Proc. 18th Annual International Symposium on Computer Architecture*, pp. 276-286, 1991.

[4] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Transactions on Computers*, vol. 37, pp. 967-979, August 1988.

[5] D. E. Culler and Arvind, "Resource Requirements of Dataflow Programs," *Proc. 15th Annual International Symposium on Computer Architecture*, pp. 141-150, 1988.

[6] J. S. Emer and R. P. Nix, "Personal Communication," June 1991.

[7] C. N. Fischer and R. J. LeBlanc, Jr., *Crafting A Compiler.* Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc., 1988.

[8] J. R. Goodman and P. J. Woest, "The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor," *Proc. 15th Annual Symposium on Computer Architecture*, pp. 422-431, June 1988.

[9] R. A. Iannucci, "Toward a Dataflow / von Neumann Hybrid Architecture," *Proc. 15th Annual International Symposium on Computer Architecture*, pp. 131-140, 1988.

[10] N. P. Jouppi and D. W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," *Proc. ASPLOS III*, pp. 272-282, 1989.

[11] D. R. Kaeli and P. G. Emma, "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns," *Proc. 18th Annual International Symposium on Computer Architecture*, pp. 34-42, 1991.

[12] K. Murakami, N. Irie, M. Kuga, and S. Tomita, "SIMP (Single Instruction Stream / Multiple Instruction Pipelining): A Novel High-Speed Single-Processor Architecture," *Proc. 16th Annual Symposium on Computer Architecture*, pp. 78-85, May 1989.

[13] A. Nicolau and J. A. Fisher, "Measuring the Parallelism Available for Very Long Instruction Word Architectures," *IEEE Transactions on Computers*, vol. C-33, pp. 968-976, November 1984.

[14] R. S. Nikhil and Arvind, "Can Dataflow Subsume von Neumann Computing?," *Proc. 16th International Symposium on Computer Architecture*, pp. 262-272, June 1989.

[15] R. R. Oehler and R. D. Groves, "IBM RISC System/6000 Processor Architecture," *IBM Journal of Research and Development*, vol. 34, pp. 23-36, January 1990.

[16] G. M. Papadopoulos and D. E. Culler, "Monsoon: An Explicit Token-Store Architecture," *Proc. 17th Annual International Symposium on Computer Architecture*, pp. 82-91, May, 1990.

[17] G. M. Papadopoulos and K. R. Traub, "Multithreading: A Revisionist View of Dataflow Architectures," *Proc. 18th Annual International Symposium on Computer Architecture*, pp. 342-351, 1991.

[18] Y. N. Patt, W. W. Hwu, and M. Shebanow, "HPS, A New Microarchitecture: Rationale and Introduction," *Proc. 18th Annual Workshop on Microprogramming*, pp. 103-108, December 1985.

[19] J. E. Smith, "A Study of Branch Prediction Strategies," *Proc. 8th International Symposium on Computer Architecture*, pp. 135-148, May 1981.

[20] J. E. Smith and A. R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors," *IEEE Transactions on Computers*, vol. 37, pp. 562-573, May 1988.

[21] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on Multiple Instruction Issue," *Proc. ASPLOS III*, pp. 290-302, 1989.

[22] G. S. Sohi and M. Franklin, "High-Bandwidth Data Memory Systems for Superscalar Processors," *Proc. ASPLOS IV*, pp. 53-62, 1991.

[23] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, pp. 25-33, January 1967.

[24] D. W. Wall, "Limits of Instruction-Level Parallelism," *Proc. ASPLOS IV*, pp. 176-188, 1991.