The PEWs Microarchitecture: Reducing Complexity Through Data Dependence based Decentralization

Narayan Ranganathan
Dept of Electrical and Computer Engineering
Clemson University
Clemson, SC 29634-0915, USA
nrangan@ces.clemson.edu

Manoj Franklin
Electrical Engineering Department
University of Maryland
College Park, MD 20742-3285, USA
manoj@eng.umd.edu

Abstract

This paper presents a microarchitecture based on exploiting the locality of data dependences for efficiently executing many instructions per cycle. The instruction window is split into multiple hardware units, and the instruction stream is distributed among them in such a way that data dependent instructions are generally allocated to the same unit. The fetch bandwidth of the processor is enhanced by doing control predictions at a level higher than that of branches, which permits fetching a large number of instructions every cycle despite the presence of taken branches in the instruction stream. The microarchitecture also off-loads most of the work involved in instruction distribution decisions to hardware that is not in the critical path of execution. It also uses a novel hardware mechanism to resolve memory references. The paper also presents detailed simulation results, which show the capability of the proposed microarchitecture to achieve high issue rates. A sensitivity study of the microarchitecture shows its issue rate to be steadily increasing as the number of hardware units is increased from 1 to 8.

Keywords: Decentralization, dynamic scheduling, hardware window, instruction-level parallelism (ILP), speculative execution.

1 Introduction

Computer architects currently face an interesting set of challenges. The steady increase in the number of transistors integrated on a chip and the clock speed has provided some excellent opportunities as well as some tough challenges. While individual gate delays and gate counts are becoming less important, wire delays are becoming a dominant factor in processor organization and design. An artifact of this is that *centralized* hardware structures having high utilization become the major bottlenecks in the processor.

In a conventional superscalar microarchitecture, the hardware typically collects decoded instructions in a centralized *instruction window*, and executes instructions as and when their source operands become available. The disadvantage of this centralized window approach is that it is not scalable: it has to contend with microarchitectural complexity and clock cycle limitations. The searching hardware needed to forward newly produced results to subsequent instructions and to identify ready-to-execute instructions from the window every cycle limits the size of the hardware window; beyond a certain length, the time required to search the window makes the system impractical [9].

This paper describes a microarchitecture called PEWs (Parallel Execution Windows) that addresses this limitation of conventional superscalar processors through the use of decentralization techniques. This microarchitecture incorporates many novel techniques as well as some existing techniques so as to obtain high performance without increasing the complexity of time-critical paths. First, it incorporates a high-bandwidth instruction fetch mechanism using high-level control flow prediction [2]. This fetch mechanism allows the processor to fetch a large number of instructions every cycle, despite taken branches in the instruction fetch stream. Second, it partitions the instruction stream among multiple *pews*[1], that execute instructions in parallel. The partitioning is done based on register data dependences. That is, two instructions that have a register data dependence between them are allocated to the same pew. Each pew has a structure similar to a conventional superscalar processor, with its own instruction window, multiple functional units, and a hardware mechanism for inter-

[1] The term *pew* (non-capitalized) denotes the hardware structure that embodies each of the parallel execution windows, and the term *PEWs* denotes the execution model that uses pews for decentralization.

pew communication. Communication between pews happens through a unidirectional ring-like network, as shown in Figure 1. A crossbar-type connection is not needed because data-dependence-based grouping directs instructions to pews in such a way as to minimize inter-pew data communication.

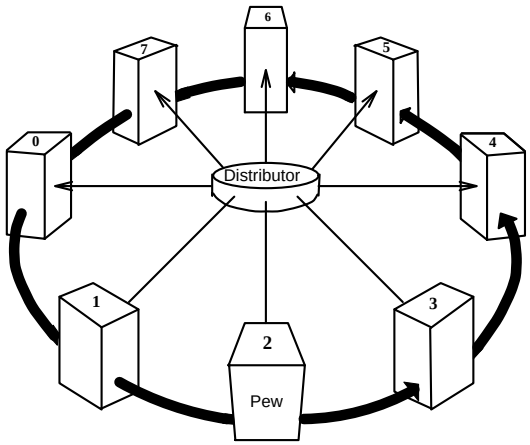

Figure 1: Block Diagram of an 8-Pew PEWs Processor

To simplify the instruction distribution process, the control flow-based representation of the incoming instruction stream is first converted to a data flow-based representation. This is done by an off-line hardware unit that determines a gist of the register data dependences among the instructions that are fetched together (in a cycle), and stores this information in a special icache. In conjunction with off-line dependence determination, some amount of static register renaming is performed, so as to reduce the amount of dynamic register renaming and the amount of write traffic to the global register file. Lastly, for carrying out dynamic reordering of memory references, the microarchitecture incorporates a decentralized hardware mechanism (called arcade) that ties well with the PEWs execution model.

The rest of this paper is organized as follows. Section 2 provides background and motivation behind decentralizing the dynamic scheduling hardware. It also provides a taxonomy of instruction window decentralization techniques. Section 3 gives the details of a microarchitectural implementation of the PEWs model. Section 4 presents simulation results of this microarchitecture. The results show that the PEWs microarchitecture is a viable option for building next generation ILP processors. Section 5 presents the conclusions.

2 Background and Motivation

Program specifications for current ISAs generally follow a *control-driven* approach. That is, control is assumed to step through instructions in a sequential order, with provisions to change the ordering by means of branch instructions. Dynamically scheduled ILP processors typically convert the total ordering implied in the program into a *partial ordering* determined by dependences on resources, control, and data. This involves identifying instructions that are are mutually resource-independent, control-independent, and data-independent. To facilitate instruction reordering and parallel execution, constraints due to resource dependences are overcome (i) by replicating resources such as the fetch unit, the decode unit, the physical registers, and the execution units (EUs), and (ii) by providing multiple banks of resources such as cache memories, as shown in Figure 2. Constraints due to control dependences are overcome by predicting the outcome of conditional branches, and carrying out speculative fetching and execution along the predicted path. Branch predictors can be scaled by making predictions at levels higher than that of branches [2]. Constraints due to data dependences can also be overcome by predicting the result of instructions, and executing data-dependent instructions with the predicted data values. Although instructions are executed out-of-order based on their dependences and associated speculations, they are committed in program order to permit recovery from incorrect speculations and to permit precise exceptions.

2.1 Need to Decentralize Dynamic Scheduler

In order to permit scaling up of a microarchitecture, resources that are in high demand should be decentralized. An inspection of the block diagram of Figure 2 shows that the important structures that remain to be decentralized are the dynamic scheduler (DS), the register rename unit, the architectural register file, and the memory address resolution unit. Incidentally, these are the most difficult parts to decentralize because of the following two reasons:

- These hardware units typically have to deal with the different kinds of dependences between the instructions decoded in a cycle, and so cannot be decentralized by mere replication.
- The DS and the memory address resolution unit need to handle not only the instructions that are

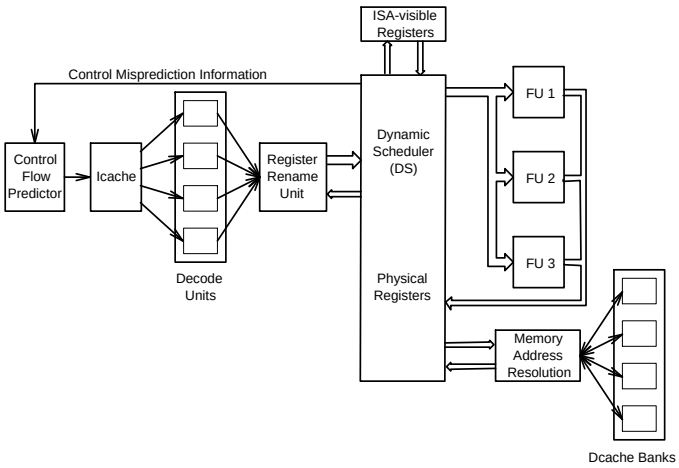

Figure 2: Organization of a Generic, Dynamically Scheduled ILP Processor

fetched in a cycle, but also the active instructions that are present in the instruction window.

A good ILP execution model should attempt to decentralize these hardware units and/or off-load as much work as possible from the critical path of instruction execution. A natural question at this point is: *on what basis should instructions be distributed among the decentralized hardware units*? The criterion used for splitting the instruction stream is very important, because an improper splitting could in fact increase communication between the hardware replicas, and degrade performance! True decentralization should aim not only to reduce the demand on each replica, but also to minimize the demand on the interconnect between the replicas by localizing a major share of the inter-instruction communication occurring in the processor to within the decentralized hardware units.

2.2 Taxonomy of Dependence-Based Decentralization

In order to place the proposed microarchitecture into a meaningful historical context, we introduce a taxonomy of dependence-based decentralization of processors. The three current approaches for grouping instructions into hardware windows have been centered around the three main constraints to execute instructions in parallel— (i) resource (EU) dependences, (ii) control dependences, and (iii) data dependences. Each of these approaches is briefly described below.

EU-Dependence-Based Decentralization

In the first approach, a hardware window is attached to each EU, so that instructions that are resource dependent on the same EU wait in the same window, and instructions that are not resource dependent on the same EU wait in different windows. Thus, instructions wait *near where they will execute*. Interestingly, the first full-blown dynamic scheduler, implemented in IBM 360/91 [13], had incorporated this approach in 1967 itself! Very recently the MIPS R10000 processor has also used this approach [8]. The advantage of EU-dependence-based partitioning is that because an instruction is waiting near its EU, it can start execution immediately after getting its source operands, provided the EU is free in that cycle. There is one main problem with EU dependence-based partitioning, however. In general, the result from an EU may be needed in any other EU, necessitating a global interconnect connecting the output of every EU to the inputs of every other EU, which does not scale well [9]. Another drawback occurs when there are multiple EUs of the same type: a ready instruction may have to wait for its EU to become free, even when another EU of the same type is free.

Control-Dependence-Based Decentralization

In the second approach, instructions are grouped based on their proximity in the dynamic instruction stream. Thus, instructions that are control-dependent on a conditional branch tend to be assigned to the hardware window to which the branch has been allocated. In this case, instructions wait *near where their control dependences will be resolved.* Examples for this approach are the *multiscalar* [5] [12], *SPSM* [1], *superthreading* [14], and *trace processors* [11] [15], where each instruction group is a task or a trace depending on whether the processor pursues multiple (independent) flows of control or not. Control-dependence-based decentralization fits well with the *control-driven* program specification typically adopted in ISAs. Because control-dependent instructions tend to be grouped together in the assembly code, no regrouping of instructions is needed at instruction commit time. Maintaining precise states at hardware window boundaries is also fairly straightforward. And, because of the nature of ordinary programs, some amount of data dependences are captured in the instruction groups.

Data-Dependence-Based Decentralization

In the third approach, instructions that are data dependent on a data-producing instruction are

most likely allocated to the hardware window to which the producer instruction has been allocated. Two instructions that do not have data dependences between them are very likely to be allocated to different windows. Here, instructions wait *near where they will get their source operands from.* The *PEWs* execution model [7], the *dependency-based* model given in [9], and the *multicluster* execution model [3] come under this category. Because data dependences dictate most of the communication occurring between instructions, data-dependence-based partitioning attempts to minimize the communication across the decentralized hardware windows. Because the instructions in a window are mostly data-dependent, there is no big need to do run-time scheduling within each window[2]. Moreover, if there is not much communication of data values between windows, then there is no need to have a global interconnect between the windows.

3 PEWs Microarchitecture

The PEWs execution model follows the data-dependence-based decentralization approach. It distributes the dynamic instruction window among multiple windows called pews. To develop a good implementation of the PEWs execution model, one must pay close attention to the following major issues:
(i) how to supply the processor with a large number of instructions every cycle?
(ii) how to distribute instructions among the pews in such a way as to minimize inter-pew communication?
(iii) how to enforce inter-pew data dependences?
(iv) how to enforce memory dependences?
(v) how to recover from incorrect speculations?
(vi) how to commit instructions in program order?
This section describes a PEWs microarchitecture that effectively addresses all of these issues. Figure 3 gives the organization of this microarchitecture. The following sub-sections describe the major blocks of this microarchitecture, and their working.

3.1 High-Bandwidth Instruction Fetching

To sustain a high issue rate, the PEWs processor must be steadily supplied with a large number of instructions per cycle. Conditional branches are a major impediment to the smooth fetching of instructions, because they are frequent, and cause control to deviate from straightline sequencing. Although high-accuracy branch prediction schemes do exist, the high frequency of branches necessitates predicting the outcome of multiple branches per cycle and fetching instructions from non-contiguous portions of code when branches are predicted to be taken [2] [10]. The PEWs microarchitecture performs this by making predictions at a level higher than that of branches.

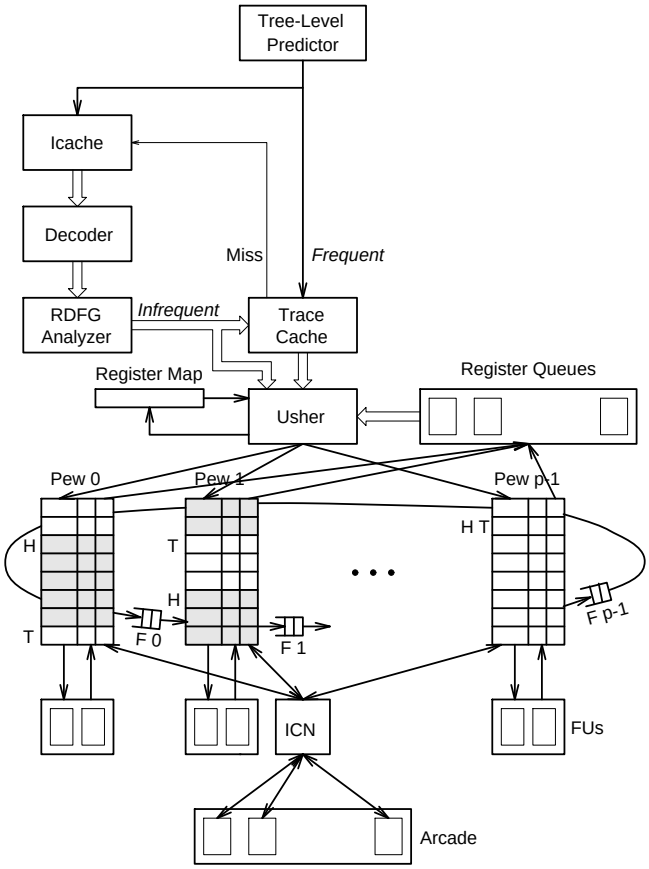

Figure 3: Organization of PEWs Microarchitecture

The predictor considers a tree-like subgraph [2] of the dynamic control flow graph as the basis of prediction. A tree of depth 4 is shown in Figure 4. This tree has 8 paths. The predictor predicts one out of these 8 paths using a 2-level predictor, and sends the attributes of the selected path to the icache and the trace cache. The trace cache caches the recently seen tree-paths, and stores the instructions of a tree-path in a contiguous manner. Normally, an instruction fetch request hits in the trace cache, and the trace cache supplies an entire tree-path in a cycle. If it misses, then the icache supplies the instructions of the path

[2] This assumes that no data value prediction is used in the processor. If data value prediction is used, then there may be merit in doing run-time scheduling within each hardware window.

to the decoder, which decodes them, and sends them to the RDFG analyzer (c.f. Section 3.1). The decoded instructions, along with the information generated by the RDFG analyzer are then written to the trace cache for future accesses. Because a tree-path contains up to 3 conditional branches, the PEWs processor can fetch about 13-15 instructions per cycle, on the average.

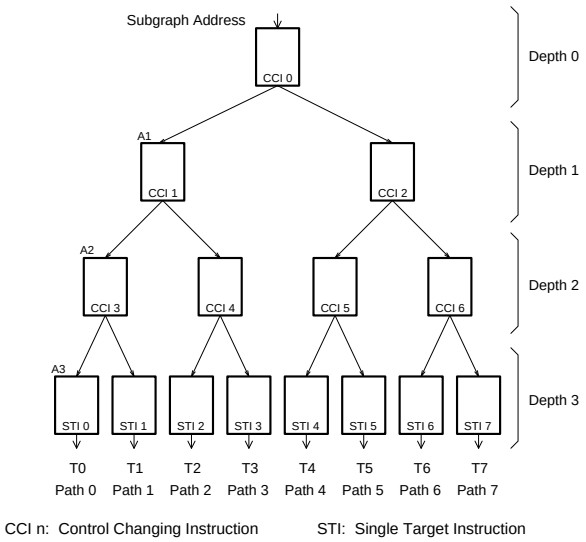

Figure 4: A Generic 8-Path Tree-like Subgraph

3.2 Preprocessing

The work involved in decoding and dispatching several instructions in parallel can be greatly simplified if register dependence checking and register renaming can be removed from the critical path of execution. To do this, the PEWs processor performs some amount of preprocessing when it encounters a tree-path for the first time.

3.2.1 Register Data-Dependency Analysis

Partitioning of instructions based on data flow is more difficult than partitioning based on either control flow or instruction type. This is because programs are generally written in control-driven form, with the result that individual strands of data-dependent instructions are often spread over a large area of code. Thus, the hardware has to first construct the data flow graph, and then do the instruction partitioning. Recognizing that the register data dependences within a tree-path are invariant (i.e., static in nature), the PEWs processor determines these dependences when the path is encountered for the first time, and caches this information for future use.

Figure 5 shows an example path and the corresponding data flow graph for the register values. The RDFG (register data flow graph) consists of several *strands* of data dependent instructions. These data dependence chains form the basis for instruction partitioning in the PEWs processor. Notice that if programs were specified in data-driven form, then data-dependence-based partitioning would have been easier. The data dependence chains in the RDFG depict the communication (through registers) occurring between the instructions in the path. Spreading a data dependence chain across multiple pews will result in inter-pew communication, which is expensive.

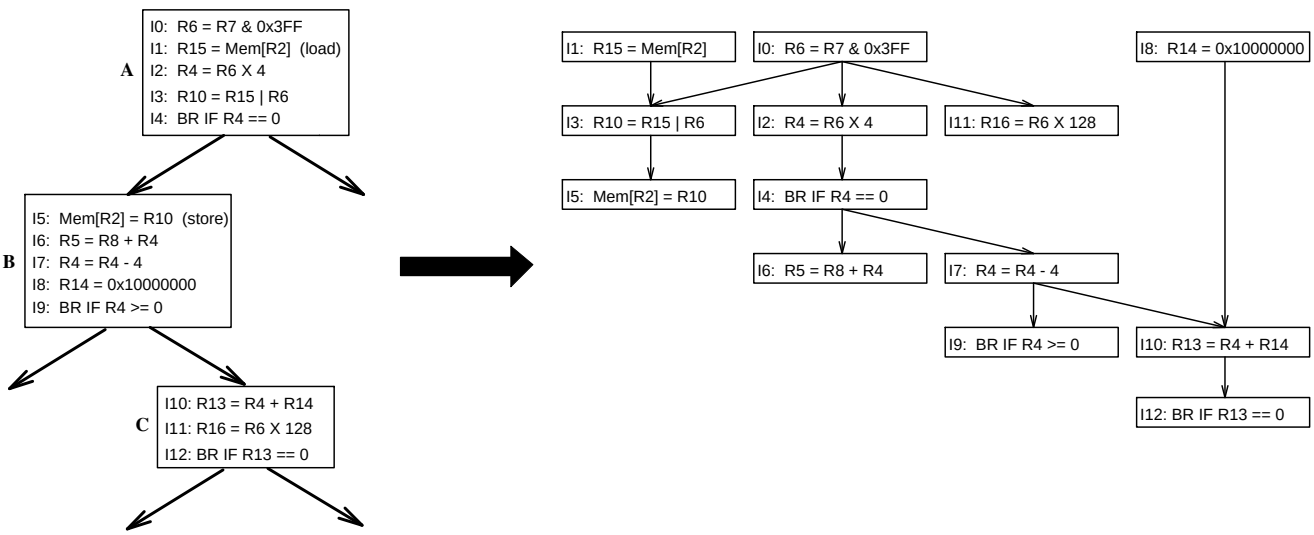

Figure 5: A Path within a Control Flow Graph, and Its Register Data Flow Graph

The PEWs microarchitecture attempts to address this problem by dispatching the data-dependence chains within a path to the same pew, as much as possible. Further, it tries to link each chain in the decoded path with the data dependence chains that are already present in the instruction window, so as to retain the complete RDFG of the instruction stream. Because the data flow graph of a program is more complex than the control flow graph, it is very difficult to achieve a partitioning that completely eliminates inter-pew communication. The PEWs microarchitecture therefore provides a ring-like interconnect for inter-pew communication.

3.2.2 Static Register Renaming

Register renaming bandwidth is unlikely to keep up with the fetch bandwidth, for PEWs processors dispatching more than a few instructions per cycle. Studies [4] [15] have shown that about half of the register instances produced in a path of 16-24 instructions are not live outside the path. This proportion naturally increases for larger paths. In the PEWs microarchitecture, this phenomenon is used to reduce the register renaming work and the register writeback traffic. Registers that are live only within a tree-path are renamed to a non-architected register. When an instruction produces the value of such a register, the value is not written to the architectural register file. A single bit tagged to each destination operand identifier in the trace cache line indicates if the specified register is from the architected or non-architected register file.

To facilitate dynamic renaming of the registers used in a tree-path and the registers updated in the tree-path, the RDFG analyzer creates two bitmaps called `use map` and `create map`, which are also stored in the trace cache. The `use map` records the architectural registers whose values are needed within the path and are produced before the path. The `create map` records the architectural registers for which new values are created in the path.

3.3 Instruction Dispatch to Decentralized Pews

Perhaps the most important functionality within the PEWs processor is the splitting of the instruction stream among the multiple pews, done by the usher hardware. In attempting to distribute the instructions fetched in a cycle, the usher should not only consider the data dependences within those instructions, but also their data dependences to the instructions that are currently present in the pews. Fortunately, this task is somewhat simplified by the data dependence information collected and stored in the trace cache by the RDFG analyzer. When the usher receives a path of decoded instructions along with the additional information, it decides the final pew placement for the instructions based on the RDFG information, the `use map`, and the current `register map`. The `register map` denotes the pews where the last instances of each architectural register (except those that are already available) will be produced. The usher decides the pew placement for the instructions at the head of each data dependence chain, and dispatches the chain to the corresponding pew. Mutually data-independent instructions are therefore very likely to be dispatched to different pews. Other, more complex, instruction distribution algorithms can be employed, and is a good subject for future research.

The usher assigns a *sequence number* to each dispatched instruction to record the instruction's place in the dynamic instruction stream. Because the original program order is destroyed when instructions are placed in the pews, the sequence numbers become useful for performing program-order related tasks, such as committing of instructions, recovering from incorrect speculations, and providing precise interrupts. Finally, the usher updates the `register map` based on the contents of the `create map` and the pew placement that it decided.

Apart from deciding the pew placement for the decoded instructions, the usher also performs register renaming. However, at run-time, a full-blown register renaming is not required because the trace cache stores partially renamed tree-paths, obtained from the RDFG analyzer. Only the registers indicated in the `use map` and the `create map` of the decoded path need to be remapped, allowing the path to go through a less complex renaming stage.

3.4 Enforcing Register Dependences

To get significant benefits from dynamic scheduling, register renaming should also be performed. The proposed PEWs microarchitecture uses a *register queue* structure to implement dynamic register renaming and speculative execution. For each architectural register the PEWs processor has a queue-like structure called a *register queue* to store multiple versions of that register[3]. A register queue entry has 4 fields—

[3] It is possible to use a pool of physical registers, along with snap-shots of register mappings, as in MIPS R10000 [8]. A potential advantage of register queues is the ease of recovery

`S#`, `P#`, `V`, and `Data`, as shown in Figure 6. The `S#` field indicates the sequence number of the instruction that produces that register instance, and the `P#` field indicates the pew where that instruction has been seated. The `V` field indicates if the value present in the `Data` field is valid.

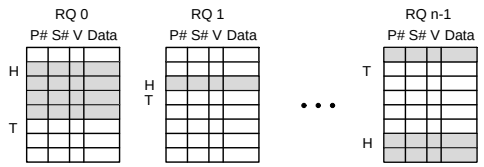

Figure 6: Register Queues

When a path and associated information are fetched from the trace cache, two things happen. First, the values of the registers specified in the path's `use map` are read from the tail entries of the appropriate register queues. At that time, the sought for register instances may either be available or yet to be produced by instructions already placed in the pews. The `P#` values of the unavailable registers are used by the usher to decide the placement of instructions among the pews. Second, once the pew placement of the instructions is determined, information relevant to the registers specified in the `create map` are entered in the tail entries of the respective register queues, and those tail pointers are advanced. The `V` bits of those entries are set to 0 to indicate that those values are not available. Notice that register values that are overwritten within a path are not allocated an entry in the register queue structure.

Intra-Pew Forwarding: When a register result is produced, it is forwarded within its pew so that subsequent instructions (in the pew) that are waiting for that value can obtain it. If it is the last update within its path to that particular architectural register, then the value is also written to the corresponding register queue entry, and the `V` bit of the entry is set to 1.

Inter-Pew Forwarding: Inter-pew forwarders, provided between adjacent pews (marked F0, F1, ... in Figure 3), forward register values from one pew to the succeeding pew, when required. The advantage of this type of forwarding is that it does not require an expensive crossbar-like connection between the pews. Note that it takes several cycles to pass a value to a distant pew, which illustrates the importance of proper instruction distribution.

and instruction retirement actions, as described in Sections 3.6 and 3.7. Investigation of different register renaming schemes for the PEWs microarchitecture is not the thrust of this paper.

3.5 Enforcing Memory Dependences

When reordering two memory operations one of which is a store, the scheduler must compare their addresses to ensure that no data dependences are violated. Typically the scheduler compares the address of every memory operation to the address of every preceding active store instruction in the processor. The PEWs processor overcomes this centralized compare by using a structure (called *arcade*), which works similar in principle to the register queues. In place of a register queue, an *arch (Address Resolution Chain)* is used for storing multiple versions of a memory address. Because memory locations are far too many to permit the use of a separate arch for every memory address, a relatively small number of arches are used, with provision to map different memory locations to an arch at different times. Each arch has an `Addr` field, which records the memory address currently mapped to it. Several arches (typically 4 - 8) are grouped into a *porch (Priority Ordered Resolution Chains)*, which keeps the arches ordered based on the sequentially earliest memory reference in its arches. The porches are interleaved based on memory addresses so that a specific porch is associated with each memory address. Each porch can accept one memory reference per cycle, and the arcade as a whole can accept as many memory requests in a cycle as the number of porches.

In contrast to a register queue, the arch records information relevant to memory reads (*i.e.*, loads) also, in addition to those relevant to memory writes (*i.e.*, stores). Accordingly, each entry of an arch has an additional bit called `L/S` bit to indicate if the reference currently mapped to it is a load or a store. The remaining fields of an arch entry are `S#`, `V`, and `Data`, as in a register queue. Figure 7 shows the block diagram of an M-porch arcade, with each porch having 4 arches. In the figure, the active portions of the arches are shaded.

It is interesting to note that the arcade is a modified form of the single-dimensional ARB (Address Resolution Buffer) given in [6]. Whereas each ARB row has as many "stages" as the instruction window size, each arch in the arcade has only a few "stages". Thus, an ARB bank can be viewed as a *large* two-dimensional *array*, whereas a porch can be viewed as a *small* two-dimensional *linked list*.

3.5.1 Execution of a Load

With arcade, load instructions are ready to execute when their address is known, even if the addresses of previous stores are unknown. When a load is executed

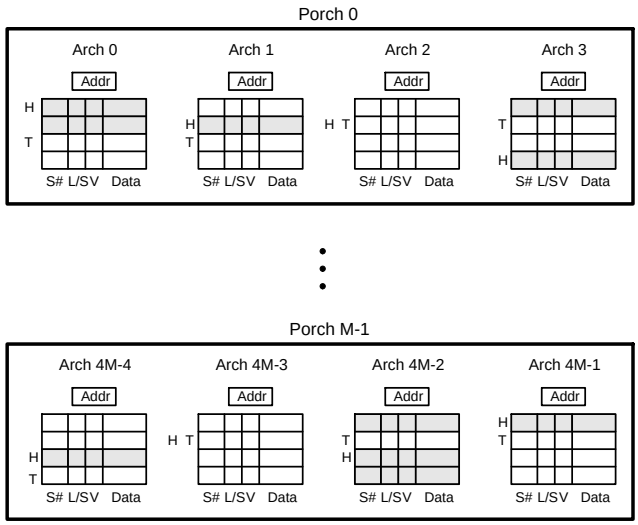

Figure 7: Block Diagram of an M-Porch Arcade

from its pew, the arcade performs two functions: (i) it finds an appropriate place in the arcade to record the load information, and (ii) it obtains the load value. First the appropriate porch is determined by taking the least significant $\log_2 M$ bits of the load address. An associative search is then done among the `Addr` field of the arches in that porch to determine if an arch has been already allotted to that address. If no matching arch is found, a free arch is allotted to the new address, and the sequence number of the load is entered in the `S#` field of the tail entry of the allotted arch; if no free arch is available, then special recovery action (c.f. Section 3.5.3) is taken. If an arch entry is already available for that address, the `S#` field of the entries in that arch are compared to determine the proper place of this load in the arch[4]. Once the proper place in the arch is determined, the sequence number of the load instruction is entered in the `S#` field of that entry, and the L/S is appropriately set to indicate a load.

The next function is to get the load value. First, the previous entry in the arch (if any) is checked to see if its `V` bit is set. If so, the value stored in its `Data` field is used as the load value. (If the `V` bit is not set, then the load instruction has to be retried later.) A copy of the value is also stored in the `Data` field of the load's entry, and the `V` bit of the load's entry is set to 1. If this is the head entry in the arch, the load request is sent to the data cache. When the required value is obtained from the data cache, a copy of the value is stored in the `Data` field of the load's entry, and the `V` bit of the load's entry is set to 1, in addition to sending the value to the load's pew.

[4]Memory addresses are generally not known at instruction decode time, and become known at different times, in no particular order. Therefore, the currently executed load may have to be inserted into the middle of an arch (this is one reason for calling it a *chain* as opposed to a *queue*). Notice that this is in contrast to the reading of a source register from the tail of the register queue at decode time.

3.5.2 Execution of a Store

Store instructions are ready to execute when their address as well as store value are known. Execution of a store also involves two functions at the arcade: (i) finding an appropriate place in the arcade to enter the store information, and (ii) checking if a succeeding load has been prematurely executed to the same address. Finding an appropriate place to enter the store information is same as that for loads. Once the proper place to insert the store is determined, the sequence number of the store instruction is entered in the `S#` field of that entry, and the L/S is set to indicate a store. The store value is also stored in the `Data` field of the store entry, and the `V` bit of the store entry is set to 1.

The next function is to check in the arch if the succeeding reference to that address is a load that has been prematurely executed. If so, recovery actions are initiated (c.f. Section 3.7).

3.5.3 Reclaiming the Arches

When a memory instruction is executed and no arch is obtained because the relevant porch is full (with other addresses), then special recovery actions are taken. This recovery action involves evicting a *lower priority* address (if any) from that porch. The porch is ordered as per priority, such that the arch whose head entry is oldest has the highest priority. The last arch of the porch is checked to see if its head entry is younger than the currently executed memory instruction. If so, the instructions including and beyond the instruction corresponding to the head entry are squashed. This action will enable that arch to be reclaimed, and be subsequently allotted to the current memory instruction. If the tail arch is found to have priority higher than this reference, then the execution of the reference is stalled until the priority of the last arch becomes lower. This event is guaranteed to occur, because the stalled reference is waiting only for previous references to be committed; once all previous references in an arch have been committed, that arch's priority becomes lower. Thus, by honoring the sequential program order in evicting arches, deadlocks are prevented.

3.6 Instruction Retirement

The PEWs processor retires (or commits) instructions in program order, because of the possibility of interrupts and incorrect speculative execution. Committing an instruction involves nothing more than advancing the head pointers of several queue structures within the processor. To commit an instruction with sequence number i, the head pointer of those register queues that have an entry with sequence number i are advanced by 1 position, if it is the last update to that register within the instruction's tree-path. For instance, if the instruction produces a new result for register `R2`, and that is the last update to `R2` in that tree-path, then the register queue corresponding to `R2` will have an entry for sequence number i. Committing the instruction then involves advancing the head pointer of register queue of `R2` by one position. If the committed instruction is a load or a store, the corresponding arch head is also advanced by one position. Also, for a store, the store value is sent from the arcade to the data cache.

3.7 Recovery from Mis-speculations

Control Mis-speculations: Any processor that executes instructions speculatively along a predicted path is bound to encounter control misspeculations, which necessitate recovery actions. Typically this involves squashing the instructions that were fetched from the incorrect path(s). Notice that these instructions might be scattered in different pews. The method employed by the PEWs processor for squashing the relevant instructions from the pews is to instruct each pew to squash all instructions that were fetched after the mispredicted branch (determined with the help of sequence numbers). All of the pews do their squashing actions in parallel by receding their tail pointers to the appropriate position. Note that this is possible because the instructions within a pew are in program order due to in-order dispatch. In addition to squashing instructions from the pews, register queue entries and arch entries corresponding to the squashed instructions need to be squashed too. This is achieved by sending the squash sequence number to each register queue and each arch so that they can squash themselves. Thus, all squashing actions are done in parallel.

Data Dependence Mis-speculations: Data dependence mis-speculations can occur because of executing loads prior to complete address disambiguation. When a store is executed, and the arcade finds that succeeding loads have been prematurely executed to that memory address, it initiates re-execution of these loads. When the loads re-execute, they forward the load values to subsequent instructions that use that value, which in turn may cause further re-executions. Notice that only the data-dependent chains get affected, and instructions that were not dependent on the load value are unaffected. Studies with an ARB [6] demonstrate that recovery actions due to dynamically unresolved references are infrequent, and therefore it is less critical to do this recovery action very fast. Alternately, if executed load instructions (in the pews) make note of the sequence number of the stores from whom they got their values, the arcade needs just broadcast (to the pews) the sequence number of the incorrect store; incorrect loads (to that address) can re-issue themselves (as in the trace processor [11]).

4 Performance Evaluation

4.1 Experimental Setup

The performance studies are conducted using the MIPS-I ISA, a representative of the class of streamlined ISAs that have emerged recently. The simulator that we developed uses the MIPS R2000-2010 functional unit latencies. It accepts executable images of programs, and simulates their execution; it is not trace-driven. All important features of the PEWs system have been incorporated in the simulator.

4.1.1 Benchmarks and Performance Metrics

For benchmarks, we use a collection of programs from the SPEC '95 suite. The programs are compiled with MIPS compilers using the optimization flags distributed in the SPEC benchmark makefiles. The benchmarks are simulated to completion or up to 500 million instructions, depending on whichever occurred first. Table 1 also gives some execution statistics, such as the number of instructions simulated, the average path (trace) length, and the path prediction accuracy. These statistics indicate that *gcc* and *go* have very poor predictability, primarily because of poor locality, leading to a large number of conflicts in the first level table of the predictor.

For measuring performance, *execution time* is the sole metric that can accurately measure the performance of an integrated software-hardware computer system. Accordingly, our simulation experiments measure the execution time in terms of the number of cycles required to execute a fixed number of instructions. While reporting the results, the execution time is ex-

Benchmark	Input File	Instr Count	Avg PL	Path Pred Accuracy
compress95	100000 q 12234	341 m	14.57	85.37%
gcc	stmt.i	150 m	13.06	81.78%
go	9stone21.in	500 m	14.29	70.17%
m88ksim	ctl.raw	500 m	12.46	96.53%
li	test.lsp	500 m	12.28	91.04%
vortex	vortex.raw	500 m	13.59	94.98%

PL – Path Length

Table 1: Benchmark Statistics

pressed in terms of instructions per cycle (IPC). Notice that the IPC figures include only the committed instructions *and do not include nops*.

4.1.2 Default Parameters for the Study

- *Dynamic Instruction Window Size:* Default size is 128 instructions.
- *Icache:* 64 Kbytes, 4-way set-associative, 1 cycle access latency, and each cache block can hold 64 MIPS instructions.
- *Trace Cache:* 128 Kbytes, 8-way set-associative, 1 cycle access latency, and each cache block can hold 16 instructions. All tree-paths starting at the same address map to the same set.
- *Data Cache:* 64 Kbytes, 4-way set-associative, 4 byte block size, 1 cycle access latency, 32-way interleaved, and non-blocking.
- *Pew Length (l):* 16 instructions.
- *Pew Issue Width (d):* 3 instructions per cycle.
- *Pew Issue Strategy:* Out-of-order execution.
- *Processor Fetch Size (f):* 16 instructions.
- *Control Flow Predictor: PAg 2-level 8-way subgraph predictor* [2], with a pattern size of 6. The SHT has 2048 entries, and is direct mapped. The PHT entries consist of 3-bit up/down saturating counters.

4.2 Experimental Results

4.2.1 Unidirectional Ring Interconnect

The first set of studies focuses on measuring how the IPC varies with the number of pews when a unidirectional ring is used to connect the pews. Figure 8 plots the IPC values obtained with the default parameters (pew length = 16; pew issue width = 3; pew issue strategy = out-of-order; fetch size = 16) for each benchmark. The values of p considered are 1, 2, 4, 8, and 12. It can be seen that the performance increases significantly as the number of pews is increased from 1 to 8. This is because the PEWs microarchitecture is better able to exploit localities of communication when instructions are spread across a moderate number of pews. As the number of pews is increased beyond 8, there is a drop in performance because some data-dependent instructions are getting allocated to distant pews, resulting in large register forwarding delays. These experimental results suggest that the algorithm used to distribute instructions is not very good for more than about 8 pews.

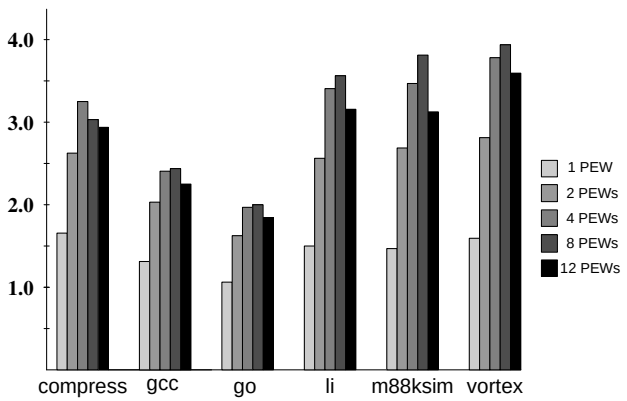

Figure 8: IPC for Varying Number of Pews: Ring Interconnect

4.2.2 Crossbar Interconnect

Figure 9 plots the IPC values obtained when the pews are connected by a 1-cycle latency crossbar. When the crossbar is used, the performance continues to improve, as p is increased beyond 8. These experimental results confirm register traffic to be the reason why this microarchitecture performs worse with large p when a unidirectional ring is used to interconnect the pews. When using a crossbar, all pews are of same proximity to each other; hence the instruction partitioning algorithm becomes straightforward. However, a crossbar does not scale well in terms of latency.

4.2.3 Trace-Cache Performance and Register Write Savings

Let us also take a brief look at the hit rates obtained for the trace cache, and the amount of register write traffic saved because of the RDFG unit renaming some of the registers to non-architected registers. Table 2 presents these results. It can be seen that with small trace caches (128 Kbytes), the number of times a request for a tree-path hits in the trace cache is high. Also, about 30% of the register write traffic has been saved by not writing back results that are updated

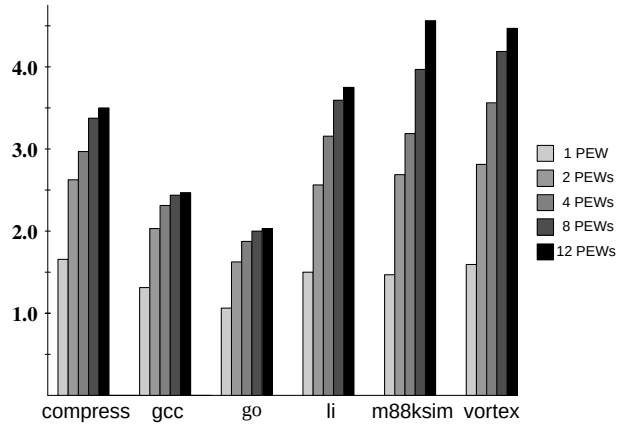

Figure 9: IPC for Varying Number of Pews: Crossbar Interconnect

within the same tree-path. As path sizes increases in the future, we can expect this percentage to increase. Table 2 also gives the percentage of memory references that resulted in the arcade initiating recovery actions because of the requested porch being full and the requested arch being full. These values, shown in the last 2 columns, indicate that arcade-initiated recovery actions are indeed very rare.

Benchmark	Trace Cache Hit Rate	Register WT Saved	Arcade-initiated Recovery due to	
			Arch full	Porch full
compress95	99.43%	19.15%	0.12%	0.00%
gcc	78.38%	30.98%	1.10%	0.00%
go	66.09%	27.70%	0.20%	0.00%
li	98.17%	32.82%	0.35%	0.00%
m88ksim	97.58%	26.45%	0.41%	0.00%
vortex	75.11%	19.34%	0.20%	0.00%

WT – Write Traffic

Table 2: Trace Cache Hit Rate, Register Write Traffic Savings, and Arcade-Initiated Recovery

4.2.4 Comparison with a Hypothetical, Centralized-Window Superscalar Processor

A hypothetical superscalar processor with a *centralized* instruction window of size 128, and capable of issuing up to 24 instructions per cycle was simulated and compared with an 8-pew PEWs processor (pew length = 16 and pew issue width = 3) having an overall instruction window of size 128 and total issue width of 24. Whereas the dynamic schedulers of the PEWs processor work on 16-instruction buffers, the dynamic scheduler of the hypothetical superscalar processor works on an 128-instruction buffer. Table 3 gives the IPC values for the two configurations. The results show that the PEWs processor is able to achieve about 76 - 94% of the IPC obtained by the hypothetical superscalar processor, with much reduced scheduler complexity [9] [11].

Benchmark	Instructions Per Cycle (IPC)		Percentage Achieved
	Hypothetical Superscalar	8-pew PEWs	
compress95	3.97	3.02	76.1%
gcc	2.72	2.44	89.7%
go	2.14	2.01	93.9%
li	4.01	3.56	88.8%
m88ksim	4.86	3.82	78.6%
vortex	4.82	3.95	82.0%

Table 3: PEWs Processor versus Hypothetical Superscalar Processor

5 Concluding Remarks

The PEWs execution model is a recently proposed paradigm for exploiting instruction-level parallelism. It is ideal for performing dynamic scheduling in a decentralized manner. The central idea behind this processing model is to split the dynamic execution window of instructions amongst smaller, parallel execution windows called pews. By keeping each pew relatively small, the circuitry needed to search it when forwarding newly produced values is greatly reduced, thus reducing the impact of dynamic scheduling on clock speed. By allocating dependent instructions to the same pew as much as possible, communication localities are exploited, thereby minimizing global communication within the processor.

This paper presented a PEWs microarchitecture that:

- incorporates control flow prediction to fetch a large number of instructions every cycle despite taken branches in the fetch stream
- performs a large part of the work involved in instruction partitioning using off-line hardware
- incorporates a decentralized hardware mechanism for reordering memory references.

We also discussed several issues related to the performance of a PEWs processor and carried out a performance evaluation of several PEWs configurations on an ensemble of well-known benchmarks. These results show that the PEWs microarchitecture is able to

achieve significantly high issue rates with decentralized hardware, and is a viable option for building next generation processors.

Acknowledgements

This work was supported by the US National Science Foundation (NSF) through a Research Initiation Award (CCR 9410706), a CAREER Award (MIP 9702569), and a research grant (CCR 9711566). The name "PEWs" was coined by Greg Kemp, who started the work on the PEWs execution model. We thank Jim Smith for his extensive comments, which have helped improve the quality of this paper.

References

[1] P. Dubey, K. O'Brien, K. M. O'Brien, and C. Barton, "Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-assisted Fine-Grained Multithreading," *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT '95)*, 1995.

[2] S. Dutta and M. Franklin, "Control Flow Prediction with Tree-like Subgraphs for Superscalar Processors," *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 258-263, 1995.

[3] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic, "The Multicluster Architecture: Reducing Cycle Time Through Partitioning," *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 149-159, 1997.

[4] M. Franklin and G. S. Sohi, "Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors," *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 236-245, 1992.

[5] M. Franklin, "The Multiscalar Architecture," *Ph.D. Thesis, Technical Report TR 1196*, Computer Sciences Department, University of Wisconsin-Madison, 1993.

[6] M. Franklin and G. S. Sohi, "ARB: A Hardware Mechanism for Dynamic Reordering of Memory Operations," *IEEE Transactions on Computers*, vol. 45, no. 5, pp. 552-571, May 1996.

[7] G. A. Kemp and M. Franklin, "PEWs: A Decentralized Dynamic Scheduler for ILP Processing," *Proceedings of the International Conference on Parallel Processing*, Vol. I, 1996.

[8] MIPS Technologies, Inc. *R10000 Microprocessor User's Manual*, Version 1.1, January 1996.

[9] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-Effective Superscalar Processors," *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[10] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," *Proceedings of the 29th International Symposium on Microarchitecture (MICRO-29)*, pp. 24-34, 1996.

[11] E. Rotenberg, Q. Jacobson, Y. Sazeides and J. E. Smith, "Trace Processors," *Proceedings of the 30th International Symposium on Microarchitecture (Micro-30)*, pp. 138-148, 1997.

[12] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar Processors," *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 414-425, 1995.

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

[14] J-Y Tsai and P-C. Yew, "The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation," *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pp. 35-46, 1996.

[15] S. Vajapeyam and T. Mitra, "Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences," *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

Narayan Ranganathan is a computer architect at Intel Inc., Oregon. He graduated from Clemson University in 1998 with an M.S. in Computer Engineering. His M.S. thesis work involved developing the PEWS microarchitecture, and studying the performance of different decentralization approaches. He obtained a B.S. in Electronics Engineering from VJTI, Bombay in 1996. His research interests are in Computer Architecture and Instruction-level Parallel Processing.

Manoj Franklin is an Assistant Professor in the Electrical Engineering Department of University of Maryland, College Park, USA, where he holds joint appointment with University of Maryland Institute for Advanced Computer Studies (UMIACS). He received the B.Sc. degree in Electronics and Communications Engineering from the University of Kerala, India, in 1984, and the M.S. and Ph.D. degrees in Computer Sciences from the University of Wisconsin-Madison, USA, in 1990 and 1993, respectively. He worked during the summer of 1990 at Cray Research Inc., Chippewa Falls, WI, USA, and the summer of 1991 at IBM T. J. Watson Research Center, NY, USA. Prior to joining University of Maryland, he was an Assistant Professor at Clemson University from 1993-1998. He received IBM Graduate Fellowship in the years 1990-1993, the NSF Research Initiation Award in 1994, and the NSF CAREER award in 1997. His primary research interests are in computer architecture, instruction-level parallel processing, digital testing, and fault-tolerant computing.