Demand-Only Broadcast: Reducing Register File and Bypass Power in Clustered Execution Cores

Mary D. Brown and Yale N. Patt

High Performance Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712-0240

TR-HPS-2004-001
May, 2004

Demand-Only Broadcast: Reducing Register File and Bypass Power in Clustered Execution Cores

Mary D. Brown Yale N. Patt

Electrical and Computer Engineering
The University of Texas at Austin
{mbrown,patt}@ece.utexas.edu

Abstract

The register file and result bypass network are large sources of power consumption in high-performance processors. This paper introduces a technique called Demand-Only Broadcast that reduces the power consumption of these structures in a clustered execution core. With this technique, an instruction's result is only broadcast to remote clusters if it is needed by dependants in those clusters. Demand-Only Broadcast was evaluated using a performance–power simulator of a high-performance clustered processor which already employs techniques for reducing register file and instruction window power. By eliminating 59% of the register file writes and data bypasses, the total processor power consumption (including the hardware needed by this mechanism) is reduced by 10%, while having less than a 1% impact on performance.

Previously published techniques for prohibiting inter-cluster result broadcast in clustered execution cores use a partitioned physical register file (vs. a replicated register file) to reduce area and access time. Register values are forwarded between clusters using dedicated hardware or copy *instructions. Copy instructions could be inserted either dynamically by hardware or statically by the compiler. These instructions lower IPC because they take window space, register file ports, and scheduling resources away from real instructions performing useful work. This paper compares Demand-Only Broadcast to a previously proposed clustered architecture which has a partitioned physical register file with the same access latency as the Demand-Only physical register file, and twice as many scheduling window entries as the Demand-Only processor. Demand-Only Broadcast reduces the number of copy instructions by a factor of 13, and results in a 10% higher IPC and 4% lower power consumption than the processor with a partitioned register file.*

1. Introduction

Many high-performance processors use large instruction windows to exploit instruction-level parallelism. Because a large window has a long access latency, the window may be partitioned into *clusters* to reduce the minimum access latency. Clustering reduces the minimum communication delays while potentially increasing the worst-case communication delays. By placing dependent instructions in the same cluster, most of the worst-case communication delays can be avoided, resulting in an overall performance improvement. Clustering the execution core does not necessarily reduce power dissipation, however, because many structures may be replicated in each cluster. This paper investigates the power and performance of clustered execution cores and introduces *Demand-Only Broadcast*, a technique for reducing the power consumption in a clustered execution core.

Some clustered microarchitectures, such as the Alpha 21264 [8], replicate the physical register file in order to reduce its access latency. By duplicating the register file and cutting the number of read ports to each copy in

half, the area, and thus the access latency, is reduced. Another advantage of duplicating the register file is that by placing one copy in each cluster in close close proximity to the cluster's functional units, the time between register file access and execution may be reduced.

In a processor with a replicated register file, an instruction's result must be broadcast to all clusters, even though it may never be needed in some clusters. With Demand-Only Broadcast, a producer instruction's result is only broadcast to the functional units and register file in a remote cluster if the instruction has a consumer in that cluster at the time that the producer's tag is broadcast to that cluster. The power consumption of the register file and bypass network can be significantly reduced by limiting the number of remote-cluster broadcasts and register file writes. If a consumer is fetched and issued to a remote cluster *after* the producer executed and the producer's result was not written to the remote cluster, then a *copy instruction* must be inserted to broadcast the producer's result to the remote cluster.

This paper evaluates the power and performance of Demand-Only Broadcast in a 4-cluster processor capable of executing up to 16 instructions per cycle. When compared to a baseline clustered processor with a replicated register file, Demand-Only Broadcast reduces the number of register file writes and tag broadcasts by 59%. While both of these models have the same register file latency, the total power consumption is reduced by 13% while having less than a 1% impact on IPC.

We also evaluate our baseline and Demand-Only models when using banked register files to reduce the access latency and power, and compare the processors with banked register files to another clustered processor that uses a partitioned register file to reduce latency and power. While holding the cycle time constant, Demand-Only Broadcast has an IPC within 1% of the IPC of the baseline processor with the banked register file, and 8% above the processor with a partitioned register file. Using Demand-Only Broadcast reduces total processor power consumption by 10% compared to just using a banked register file, and the power consumption with Demand-Only Broadcast is 4% lower than that of the processor with a partitioned register file.

Section 2 discusses related clustering techniques which limit result broadcast. Section 3 explains the baseline processor which is used to evaluate Demand-Only Broadcast, and Section 4 describes its implementation. Sections 5 and 6 explain the experimental framework in which the processor models are evaluated and the results, and Section 7 concludes.

2. Related Work

Several processor paradigms have sought to decentralize the execution core and limit the communication between clusters. We will limit our discussion to microarchitectures which implement sequential ISAs [15].

2.1. Multiscalar Processors

In the Multiscalar processing paradigm [17], a program's instruction stream is divided into tasks which are executed concurrently on several processing units. Each processing unit contains its own physical register file. Because there may be data dependences between the tasks, the processor must support register result forwarding and memory dependence detection between the processing units. Because each task is a contiguous portion of a program's dynamic instruction stream, only the live-out register values from a task must be forwarded to successive tasks executing on other processing units. The compiler can identify the instructions that may produce live-out register values, and inserts instructions called *release instructions* into the code indicating that the values may be forwarded to other processing units.

2.2. Clustered Microarchitectures

There are several microarchitectures which use a centralized instruction fetch unit but send instructions to one of several execution clusters based on data dependencies. One example which uses a centralized register file is the PEWs [10] (Parallel Execution Windows) microarchitecture. A PEWs processor contains several execution windows that are connected in a ring. Buffers between each adjacent pew hold register values that need to be broadcast to other pews. Only one value can be forwarded between adjacent pews per cycle. Demand-Only Broadcast, however, does not require forwarding buffers and does not place any restrictions on the number of values forwarded per cycle.

In the Multicluster Architecture [6], the physical register file, scheduling window, and functional units are partitioned into clusters. Each cluster is assigned a subset of the architectural registers. If an instruction's register operands must be read from or written to more than one cluster, copies of the instruction must be inserted into more than one cluster. These extra instructions must contend with regular instructions for scheduling window write ports, register file ports, execution cycles, and space within the instruction window. Hence they may lower IPC, although the Multicluster paradigm benefits from a higher clock frequency compared to a centralized core.

In the architecture described by Canal, Parcerisa, and González [5, 13], each cluster contains a partition of the physical register file and scheduling window, as well as a subset of the functional units. While dependent instructions within the same cluster can execute in back-to-back cycles, inter-cluster forwarding takes two or more cycles. Instructions write their register results only to the partition of the physical register file in their local cluster. If an instruction needs a source operand that resides in a remote cluster, a special *copy* instruction must be inserted into the remote cluster. Only copy instructions may forward register values between clusters. By limiting the number of copy instructions that can be executed in a cycle, the number of register file write ports and global

bypass paths can be reduced. This will reduce the register file and scheduling window access times and increase the clock frequency. Furthermore, since the entire register file is not replicated across clusters, each partition can have fewer entries than if the entire register file were replicated, which further reduces the register file access time. However, as with the Multicluster paradigm, the copy instructions may lower IPC.

The clustered architecture described by Zyuban and Kogge [21] also uses a partitioned physical register file. However, rather than using copy instructions to broadcast results, dedicated hardware is used to support copy operations. Each cluster has a scheduling window for determining when values are ready to be copied to another cluster. This window is similar to a traditional scheduling window except that it may be smaller, and only 1 source is needed per operation, while real instructions may have 2 (or more) operands. Each cluster also has a CAM to hold results that were broadcast from other clusters, rather than using extra physical register file entries as in the architecture described by Canal, Parcerisa, and González. Their technique assumes that a regular instruction can wake up, arbitrate for execution, and broadcast its tag to a copy operation which then wakes up, arbitrates for execution, and broadcasts its tag to a dependent instruction in another cluster, all within one clock cycle. Demand-Only Broadcast does not add any additional latency to the scheduling logic.

3. Processor Overview

The baseline processor used for this paper is a 15-stage superscalar processor with an execution core partitioned into 4 clusters, each capable of executing 4 instructions per cycle. The pipeline is shown in Figure 1. The dark lines separate the in-order and out-of-order stages of the pipeline, and the shaded stages denote the operations that are local to each cluster. Each cluster holds one fourth of the scheduling window entries, and like the Alpha 21264 [8], each cluster contains a copy of the physical register file. Figure 2(a) shows an overview of the execution core. Instructions are fetched and decoded in the first 4 cycles. In the next 6 cycles, instructions are renamed, steered to a cluster, and issued[1]. The renaming logic maps an instruction's architectural source registers to physical registers. After the instruction's source registers have been renamed, the instruction is steered to a cluster and its destination register is assigned a physical register. The steering mechanism will be described in more detail in Section 3.3. After instructions are steered to a particular cluster, they are issued (i.e. inserted into the cluster's scheduling window). The issue operation is discussed in Section 3.2. After an instruction becomes ready and is selected for execution, it reads the register file and then executes. The scheduling logic is discussed in Section 3.1.

Figure 2(b) shows the contents of one cluster. Each cluster contains a Busy-Bit Table [19], the scheduling window and scheduling logic, a copy of the register file, four functional units, and bypass logic for both data and tags. While our simulation model assumes all-purpose functional units, Demand-Only broadcast can be

[1]In this paper, *issue* means insert into a scheduling window.

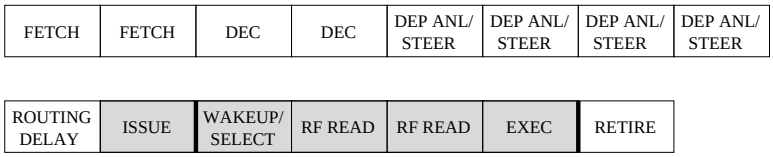
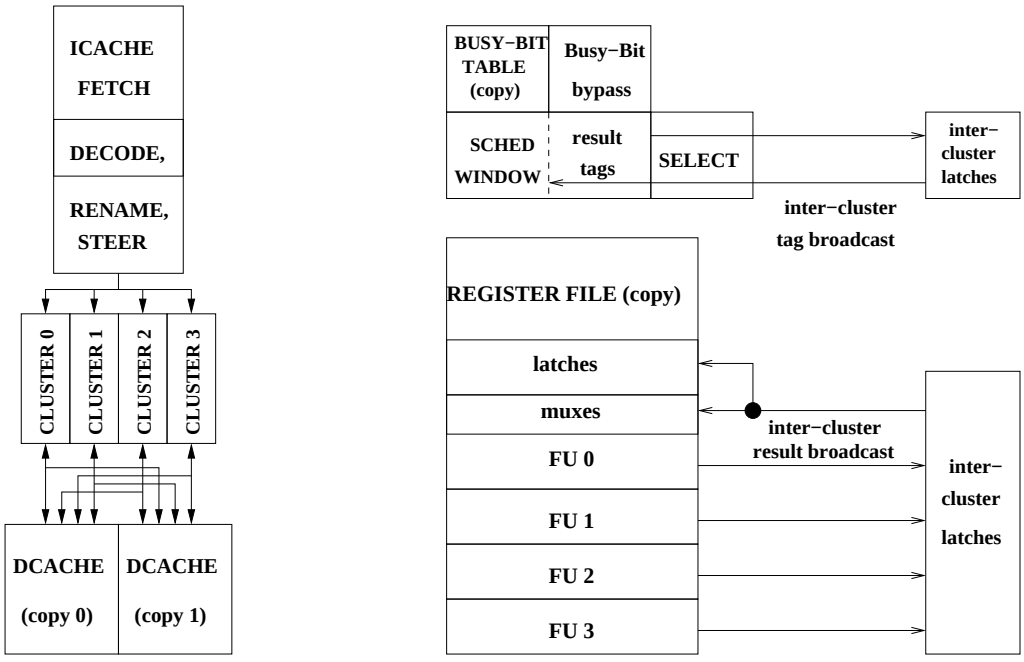


Figure 1. pipeline

generalized for processors with special-purpose functional units.

Because it takes one cycle to forward data across one cluster, there will be 1 cycle bubble between the execution of an instruction in cluster 0 and a dependant in cluster 1; there will be 2 bubbles between the execution of an instruction in cluster 0 and a dependant in cluster 2; and so on. The data cache is replicated in order to reduce the number of read ports and hence load access latency. Stores must write data to both copies, but loads read from only the closest cache.



(a) Execution Core (b) One Cluster

Figure 2. Execution Core and Cluster Overview

3.1. Scheduling Operation

The scheduling logic uses conventional wakeup and select logic described by Palacharla, Jouppi, and Smith [12]. The scheduling window holds instructions that are waiting to execute. Each entry holds the physical register numbers (SRC TAG) and Ready (R) bits for an instruction's source operands. A portion of one entry is shown in

Figure 3. An instruction's Ready bits are set when its source operands' tags have been broadcast, and it requests execution after all of its Ready bits are set. The Destination Tag Array holds the destination physical register number for each instruction in the window. When an instruction is selected for execution, the select logic accesses the portion of the window holding the destination tags to broadcast the instruction's destination tag to the scheduling window. The instruction is deallocated from the scheduling window after it executes.

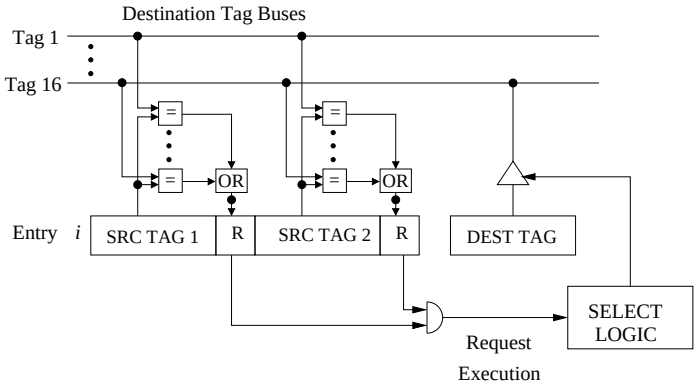


Figure 3. One Scheduling Window Entry

The number of entries in the scheduling window is half the number of entries in the instruction window (i.e. the maximum number of issued but non-retired instructions). Our simulations showed that using more entries (up to as many as the instruction window will allow) improved IPC by less than 2%, but the IPC started to drop rapidly when the size was further decreased. To save power, the scheduling window is non-compacting – that is, instructions stay in the same location for the duration of the time they are in the window. While oldest-first selection logic is not used, previous work has shown that the selection priority has little effect on IPC [4]. Since an instruction's consumers may reside in any cluster, it broadcasts its destination tag and result to its local cluster in addition to all other clusters. Because instructions are scheduled for execution several cycles before they execute, their tags are broadcast several cycles before their data is broadcast.

3.2. Instruction Issue

When an instruction is first placed into the scheduling window, it must know if its source operands are ready. The Busy-Bit Table (BBT) is used to determine this. This table, which is indexed by physical register number, indicates which instructions have already broadcast their destination tags to the local cluster. When an instruction is issued, it reads the BBT entries corresponding to the physical registers of its source operands, as well as the tag buses. If the BBT entry of a source operand is set or its tag is broadcast in that cycle, then the instruction sets

the Ready bit of that source operand. If the BBT entry is clear and the tag is not broadcast in that cycle, then the Ready bit of that source operand is not set.

Each cluster has its own copy of the BBT because different clusters will receive an instruction's tag broadcast in different cycles. The BBT has two read ports for every instruction that can be issued to the local scheduling window in a given cycle. Since up to 16 instructions may broadcast their tag in a given cycle, 16 bits of the table may be set in a given cycle. Since up to 16 new issued instructions are assigned physical destination registers each cycle, 16 bits may be cleared in a cycle. Any arbitrary number of bits may be cleared in the event of a branch misprediction. For the purposes of measuring the power consumption of the BBT, we assume just two additional wordlines are needed per bit to support the setting and clearing operations. The physical register numbers for each instruction are decoded and ORed together before updating the BBT.

Since up to four instructions may execute in a given cluster in one cycle, each copy of the physical register file has 8 read ports (two per instruction). In our baseline, the register file has 16 write ports, although Section 3.4 will discuss how the number of write wordlines is reduced.

3.3. Instruction Steering

The performance of a clustered processor is sensitive to the steering mechanism used [2, 13]. Most steering algorithms try to address two adverse goals: (1) minimizing inter-cluster communication and (2) load balancing in order to effectively use all of the processor's resources. We have experimented with using combinations of several heuristics including Modulo-N [2, 5], dependence-based [12], predicted Last-Source-Ready [18], and the DCOUNT threshold [13]. With the Modulo-N heuristic, the first N instructions in program order are assigned to cluster 0, the next N instructions are assigned to cluster 1, and so on. With dependence-based steering, instructions are steered to the clusters which hold their source operands. Predicted Last-Source-Ready is a form of dependence-based steering in which, if an instruction has two source operands, it is steered to the cluster of the source operand that is predicted to be available last. The DCOUNT metric is a function of a processor's load imbalance. If this metric exceeds a threshold, then a load-balancing heuristic overrides a dependence-based heuristic.

Our baseline processor limits the number of instructions that can be written into a given cluster in one cycle. This steering limitation reduces the number of write ports to the scheduling window (thus reducing scheduling latency and power) and also serves to load-balance the clusters. When limiting the number of write ports to four or six per cluster, additional load-balancing heuristics such as DCOUNT did not improve IPC for either our baseline processor or the other processors discussed in Section 5.

For this paper, we use a dependence-based steering algorithm. An instruction's cluster preference is the cluster

which holds its register source operand. If an instruction has two operands, the first is used by default.[2] If it has no source operands, its cluster preference is assigned according to a Modulo-4 heuristic. This algorithm, while not optimal, performed the best of all of the viable steering heuristics we studied on all of the models discussed in Section 5. The cluster containing the source operand is identified by an additional 2 bits in the Register Alias Table of the processor without the banked register file, and by the most significant two bits of the physical register number in the processor with the banked register file. The steering logic also keeps track of the number of free scheduling window ports and entries (and physical register file entries in the case of the banked register file). If any of these resources are not available in the desired cluster, the instruction is steered to the closest cluster.

We also examined the effects of stalling issue if an instruction could not be placed with its source operand because of one of the necessary resources (i.e. a port or a window slot) was unavailable. Stalling issue because of unavailable scheduling window write ports severely hurt IPC in all of the SPECint2000 benchmarks. Stalling because of a full window or register file helped slightly in a few benchmarks, but for the average of the suite, IPC was lowered. Hence our steering logic does not stall issue unless an instruction cannot be placed in any cluster.

3.4. A Banked Register File

In this paper, we evaluate Demand-Only Broadcast using the baseline machine described above, as well as with a machine with a banked register file. In both models, the register file is replicated in all four clusters.

For this paper, we have chosen just one method of register file banking, although Demand-Only Broadcast can be used with other types of banked register files as well[1]. The method we use is similar to the Register Write Specialization first described by Seznec et al [16]. This paper will provide an overview of the method.

Although potentially 16 instructions can write back to each register file in a given cycle, the register file is divided into four banks in order to reduce the number of write ports. All instructions residing in a given cluster will write to the same bank, and no other instructions will write to that bank. For example, in our baseline configuration there are 512 physical registers. All instructions in the first cluster are assigned a physical register number between 0 and 127; all instructions in the second cluster are assigned a physical register number between 128 and 255, and so on. Because only four instructions from each cluster may execute in a cycle, only four write wordlines will be needed for each bank. All four banks are stacked vertically, which means that the width of the register file is still determined by the total number of write and read ports: 16 and 8, respectively. The height of the banked register file is reduced because each bit cell has only 4 write wordlines rather than the 16 that are needed in

[2]For instructions with two source operands, random operand selection and Last-Source-Ready prediction did not significantly improve IPC. Knowledge of whether or not a source operand was “Ready” would further complicate the steering logic, and it would require adding additional ports to the BBTs.

the unified register file. As in the baseline, instructions must still write their results to all four copies of the register file. Using a banked register file adds an additional constraint on cluster assignment: if all physical registers in a particular bank have been allocated, no instructions may be steered to that cluster, even if there is room in the scheduling window. Our simulations show that this style of register file banking affects IPC by less than 1% while greatly reducing the power consumption and/or access latency.

4. Demand-Only Broadcast Implementation

When using Demand-Only Broadcast, an instruction does not broadcast its result to the register file and functional units in another cluster unless that cluster holds a consumer when the instruction's tag is broadcast. Each cluster keeps track of which physical registers are needed by instructions within the cluster. The Busy-Bit Table is extended to hold this information. Rather than just 1 bit for each entry, there are two: the "Broadcast bit" indicates if the tag has been broadcast, and the "Use bit" indicates if there are any instructions within the cluster requiring that physical register.

When an instruction is first placed into the cluster, it reads the BBT entries corresponding to its source operands as the baseline does. The Broadcast bits indicate if the source operands' tags have been broadcast, and the Use bits indicate if there are older instructions in the same cluster requiring the results of those instructions. The instruction sets the Use bits of those entries as well as the Use bit for the entry of its own destination physical register number.

When an instruction broadcasts its tag to a cluster, it sets the Broadcast bit for that instruction's destination register, just as in the baseline machine. Additionally, it reads out the value of the Use bit. If the Use bit is set, the instruction's result will be broadcast to the register file and functional units in this cluster. If the Use bit is not set, the broadcast will be blocked.

Normally, the instruction's data would be broadcast N cycles after its tag is broadcast (assuming N is the number of pipeline stages between the scheduling logic and the final execution stage). When the Use bit is read, it will enable the latch for the data result bus N cycles later. Because the value of N is generally at least as long as the minimum number of cycles for the register file access plus execution (5 cycles in the Intel Pentium 4 [9]), there is plenty of time to set the controls to gate the data broadcast and prevent the register file write.

Table 1 gives an example in which an instruction *A* in Cluster 0 produces a value needed by instruction *B*, which is issued to Cluster 3. In this example, an instruction's result is broadcast 2 cycles after its destination tag. BBT-0[A] refers to the entry of the Busy-Bit Table in Cluster 0 corresponding to A's destination register, and BBT-3[A] refers to A's entry in the Busy-Bit Table in Cluster 3. In cycle 2, instruction *B* is issued to Cluster 3, and it reads and updates the BBT entry for instruction *A*, and it sets the Use bit for its own entry. In cycle 3, *A*'s tag is broadcast to Cluster 3. By this time, BBT-3[A].use has been set, so *A*'s result will be broadcast 2 cycles later.

Cycle	Initial state: *A* is in Cluster 0's scheduling window. BBT-0[A].use = 1.
0	*A* is selected and broadcasts tag to Cluster 0. Set BBT-0[A].broadcast = 1.
2	*A*'s data is broadcast to Cluster 0. *B* is issued to Cluster 3. Set BBT-3[A].use = 1.
3	*A*'s tag is broadcast to Cluster 3. Set BBT-3[A].broadcast to 1. BBT-3[A].use was 1, so don't block broadcast. *B* wakes up.
4	*B* is selected.

Table 1. Timing for an inter-cluster broadcast.

4.1. Copy Instructions

When an instruction is first issued, the instruction producing its source operand may have already executed, and failed to write its result to the register file. This would have happened in the example above if instruction *B* were issued after cycle 3. In this situation, a *copy instruction* will be required to re-broadcast the result. The broadcast instruction will be inserted into the cluster that produced the source operand (although it could actually be inserted into any cluster that didn't block the broadcast). The copy instruction will read the register file and re-broadcast the physical register destination tag and the data, similar to a MOVE instruction with the same physical source and destination register.

In order to detect if a copy instruction is needed, when an instruction is first issued and reads the BBT entry of its source operand, it must read out the old value before it is set, like a scoreboard. If the Use bit is clear and the Broadcast bit is set, then a copy instruction must be inserted.

Insertion of Copy Instructions

Each cluster creates a bit-vector specifying which physical registers require copy instructions to re-broadcast the data. All instructions issued to a cluster may set bits of its bit-vector. If an instruction reads a 1 for the Broadcast bit and a 0 for the Use bit of one of its source operands, the corresponding bit of the bit-vector is set.

The bit vectors from all four clusters are ORed together to form the *Copy Request Vector*. This vector specifies all physical registers requiring a copy instruction. The Copy Request Vector is later used by the steering logic to insert copy instructions. Assuming all instructions could have at most 2 source operands, up to 32 bits of this vector could be set each cycle if 16 instructions are issued per cycle. A priority circuit is used to pick up to four physical registers per cluster for which to create copy instructions. The steering logic will then clear the selected bits of this vector and insert copy instructions for the selected physical registers.

Copy instructions are not inserted until at least five cycles after the consumer instructions requiring the re-broadcast have been issued. This 5-cycle delay is due partially to the fact that the steering logic may have already

begun to steer instructions that will be issued within the next 3 cycles, and we assume there is a 2-cycle delay between the issue logic and the steering logic.

The example in Table 2 illustrates the copy insertion timing. When instruction *A* broadcasts its tag to Cluster 3, the Use bit of its BBT entry is clear, so its data broadcast will be blocked 2 cycles later. When instruction *B* is issued to this cluster and reads the BBT, it must request a copy instruction because the Use bit of A's BBT entry was clear while its Broadcast bit was set. Instruction B then resets the Broadcast bit and sets the Use bit of this entry. By cycle 6, a bit of the Copy Request Vector corresponding to A's destination has been set and the steering logic inserts a copy instruction. In cycle 9, the copy instruction is issued into cluster 0.

Cycle	Initial: *A* is in Cluster 0. BBT-0[A].use is 1, BBT-3[A].use is 0.
0	*A* is selected and broadcasts tag to Cluster 0. BBT-0[A].broadcast = 1.
3	*A*'s tag broadcast to Cluster 3. Set BBT-3[A].broadcast. Block result broadcast (2 cycles later).
4	*B* is issued to cluster 3. BBT-3[A].use = 1 and BBT-3[A].broadcast = 0. Request copy.
6	CRV[A] is set.
9	*Copy-A* is issued (automatically awake) and selected for execution.
12	*Copy-A* broadcasts tag in Cluster 3; *B* wakes up. Set BBT-3[A].broadcast.
13	*B* is selected and broadcasts its tag to Cluster 3.

Table 2. Timing for Copy Instruction Insertion.

Not only do copy instructions delay the execution of their dependants, but they may take resources away from real instructions performing useful work. They occupy issue ports, possibly causing instructions in the renaming stage to be stalled or steered to an undesired cluster. They will also occupy space in the scheduling window before they are executed, although they do not remain in the window long because they are already "Ready" when they are placed in the window. They may also prevent a real instruction from being selected for execution as soon as possible, since copy instructions must be selected and access the physical register file like regular instructions. This extra demand on the hardware resources may lower IPC and consume power. However, because copy instructions are only inserted if an instruction's source operand was steered to a different cluster *and* that operand was already broadcast *and* it was not written to the local physical register file, copy instructions are rarely needed and impact the IPC by less than 1%. Section 6 will show power and performance results.

5. Experimental Framework

We have measured the IPC and per-cycle power consumption for five processor models: the baseline processor with a replicated, but not banked, register file (BASE-UNI), the baseline processor with a replicated, banked register

file (BASE-BANKED), processors which use Demand-Only broadcast with and without banked register files (DO-BANKED and DO-UNI) and a model similar to the Parcerisa and González [13] paradigm which uses a partitioned register file (PART). Section 5.1 will explain the model with the partitioned register file, and Section 5.2 will discuss our power and performance simulators and provide more details about the machine models.

5.1. Partitioned Register File Model

The machine model with a partitioned register file is a 16-wide, 4-clustered microarchitecture just like the Baseline. The fundamental difference is that the physical register file is partitioned rather than completely replicated, with each cluster holding one fourth of the entries. When instructions execute, they broadcast their result only to the cluster in which they reside. Likewise, when instructions are selected for execution, their destination tag is only broadcast to the local cluster.

Copy instructions must be used to forward data from one cluster to another. Copy instructions are the only instructions that broadcast tags and data from one cluster to another. By limiting the number of copy instructions that can be executed, the number of register file write ports and data and tag buses can be reduced. Excluding copy instructions, each cluster needs 4 tag buses and write ports, assuming only 4 instructions per cluster finish execution per cycle. By assuming each cluster can execute at most 1 copy to each remote cluster per cycle, each cluster will need a total of 7 write ports and buses: 4 for regular instructions and 3 for copy instructions. Like the results reported by Parcerisa et al. [14], our simulations showed that adding more bypass buses and ports did not significantly help IPC. However, further reduction in the number of ports would complicate the scheduling logic because multiple clusters would have to arbitrate for the ports and buses.

When a copy instruction is executed, the value it is copying will be available in two physical registers in different clusters. Since an architectural register may be valid in more than one cluster, the Register Alias Table keeps track of up to four mappings for each architectural register. When an instruction retires, all valid physical register entries belonging to the previous instance of the instruction's architectural destination register must be deallocated.

In this paradigm, instructions do not need to read the BBT before determining if a copy instruction must be inserted to receive a source operand. Instructions determine if a copy instruction is needed after they have been assigned to a cluster. If an instruction is steered to a cluster which does not have a valid physical register mapping for one of its source operands, then a copy instruction is needed. In order to avoid any bias towards the Demand-Only model, we will assume that the PART model can issue a copy instruction in the same cycle as the instruction requiring the copy. Note that this is an aggressive assumption because according to the steering algorithm used, the subsequent instructions cannot be assigned to clusters until after the copy instruction and instruction requiring

the copy have been assigned to clusters. When the instruction is steered and updates its RAT entry, the RAT entry of the register being copied is also updated to indicate that it has a valid mapping in the cluster to which the dependent instruction was steered.

Because values may reside in more than one register file partition, each partition should have more than one fourth of the physical register file entries that the Baseline model has in order to prevent the processor from running out of physical register entries too frequently. We chose to use physical register file partitions with 224 entries. This number was selected for two reasons: (1) it is scaled linearly from the configuration used by Parcerisa et al. [14] (the 4-cluster model has 1.74 times as many entries as the 1-cluster model); (2) further decrease in the size caused an IPC degradation in a few benchmarks, while further increase did not noticeably affect IPC.

The scheduling window in this model is smaller than in the other models because there are only 7 tag buses per window rather than 16. The number of scheduling window entries was increased from 64 per cluster to 96 per cluster to account for the copy instructions. While the smaller window may allow the clock frequency to be increased, we will assume the clock frequency remains constant in order to make a fair comarison of the power consumption.

5.2. Power and Performance Models

Our simulator is a cycle-accurate, execution-driven processor which models mispredicted-path effects and executes the Alpha ISA. Our power model is based on the Wattch framework [3], although it has been heavily modified to work with our processor simulator and accurately represent our processor models. The functions for estimating the power of the basic processor building blocks (arrays, CAMS, some combinational logic and wires, and clock distribution) are taken from Wattch, although the method of access counting has been modified. This section discusses the major changes to Wattch.

First, most of the access counters have been changed from those present in the original Wattch framework. Our model distinguishes between different types of accesses to many structures. For example, data cache reads and writes do not consume equal amounts of power. The largest difference is due to the fact that the cache is duplicated in order to reduce the access latency by halving the number of read ports to each copy. A write, from either a store instruction or a cache-line fill, must update both copies of the cache.

In the register file, writes also have a disproportionate power dissipation compared to reads [20]. The primary discrepancy is that conventional register file bit cells have two bit lines for each write port and one bit line for each read port [7]. As a result, we model reads and writes as different types of accesses.

We have made some assumptions about the floorplanning of the execution core in order to model the power

dissipation of the result bypass network. Within each cluster, all functional units are stacked vertically as shown in Figure 2 so that the data bitlines are interleaved. The register file sits directly above the functional units, muxes, and the latches which hold data being read to and written from the register file, in addition to the data that was broadcast from other clusters. The width of each cluster is a function of both functional unit area estimates [11] as well as the maximum number of bitlines at any point in the datapath for wide-issue clusters. We conservatively assume that this width is constrained by width of the register file. In the baseline configuration, the result bus from each functional unit runs vertically within its own cluster to the register file write latch, as well as horizontally to the other clusters and then vertically across all other stacks as well. In the model with the partitioned register file, the result buses run to only the local physical register file and bypass muxes.

Some of the additional units in our power model not present in Wattch include a 32-entry Memory Request Buffer that holds memory requests that miss in the L1 instruction and data caches (each entry supporting up to 4 piggy-backed instructions), multi-ported instruction and Level-2 caches (as well as a duplicated data cache), and logic for inserting copy instructions. The processor configurations that are the same for all of our models are listed in Table 3, and those that depend on the model are listed in Table 4. All of the units listed, as well as 15 pipeline stages, were modeled with Wattch. A conditional clocking style similar to that of CC3 in Wattch is used: an array's power dissipation scales linearly with the number of ports used, except that all units dissipate *at least* 10% of their maximum every cycle, even when they are not accessed *or* fewer than 10% of the ports are accessed.

Instruction Cache	64KB 4-way set associative, 64B line size 2 ports, 2-cycle directory and data access,
Branch Predictor	hybrid 64K-entry gshare/PAs, 4096-entry 4-way BTB, 32-entry RAS
Decode, Rename, Steer	16 instructions per cycle, 6 cycles
Issue and Execution Width	16 general-purpose functional units
Data Cache	2 copies, 64KB, 4-way set associative, 64B line size 2 read ports, 2 write ports (per copy). 3-cycle loads
Instruction Window	512 instructions in-flight
Unified L2 Cache	1MB, 8-way, 64B lines, 10-cycle access 2 banks each with 1 read, 1 write port, contention is modeled
Main Memory	32 banks, 100 cycles access (minimum)

Table 3. Common Processor Configurations

6. Results

We have evaluated the five models with both 4 and 6 issue ports per cluster on the SPECint2000 benchmarks using modified input sets to reduce simulation time. The average power estimates shown in this section are based on "per-cycle" power estimates, although not all configurations may run with the same cycle time. Because the BASE-UNI and DO-UNI models have larger register files, they may run with at a lower frequency than the

	BASE-UNI	BASE-BANKED	DO-UNI	DO-BANKED	PART
PHYS. REG. FILE					
entries, per cluster	512	512	512	512	384
entries, entire core	512	512	512	512	1536
write wordlines	16	4	16	4	7
write bitlines (dual rail)	16	16	16	16	7
read ports	8	8	8	8	8
SCHED WINDOW					
num entries (per cluster)	64	64	64	64	128
tag buses	16	16	16	16	7
source tag size (bits)	9	9	9	9	9
RAT entry size (in bits)	11	9	11	9	40
BBT					
num entries, per cluster	512	512	512	512	384
entry size (bits)	1	1	2	2	1
num decoders	40	40	40	40	31

Table 4. Model-Specific Processor Configurations (assuming 4 issue ports per cluster)

other models (assuming a non-pipelined register file). In addition, the configurations with 4 issue ports have different scheduling window latencies than the configurations with 6 ports so they may run at different frequencies. However, we show all per-cycle results within the same graph to save space.

The average per-cycle power dissipation for each model relative to the Baseline is shown in Figure 4. The numbers at the top of each bar are the harmonic means of the IPC for that model. Each bar shows the contribution of each of the processor units to the total processor power consumption. The components that are not directly affected by our technique fall under the "other" category, although they may be indirectly affected by modifications in the program behavior. The components are shown in the order listed in the graph's legend. The bottom two categories measure all dynamic power consumption due to result broadcasts and register file writes. The bottom category, "Network Broadcast", is the power consumption of the horizontal data buses running between the clusters. In the Demand-Only models, *all* results are broadcast across the entire network bypass, although the PART model only broadcasts the results of copy instructions across the network. The next component, "Cluster Broadcast", is the power consumption for broadcasting a result within a stack of functional units and writing this result back to the register file. The power dissipation in this category is dominated by the register file write, not the bypass buses. In the BASE-UNI model, the register file write accounts for about 85% of this component. The "PRF Read" component includes all dynamic power consumption for the register file reads, in addition to the power consumption when the register file is not accessed at all (i.e. the "turnoff" power). The power consumption of the logic for inserting copy instructions was less than 0.2% of the total power consumption. Almost all of the BBT power dissipation is from the decoders, not the BBT array itself, which is just a few bit-vectors. The PART model had a higher power dissipation than the DO-BANKED model because the rename logic had to keep track of

four mappings per architectural register. It does benefit from having far fewer scheduling tag broadcasts, though. For the 4-port configurations, the power consumption for the DO-BANKED model is 10% lower than that of the BASE-BANKED model, and 4% lower than that of the PART model. The power consumption of the DO-UNI model is 13% lower than that of the BASE-UNI model.

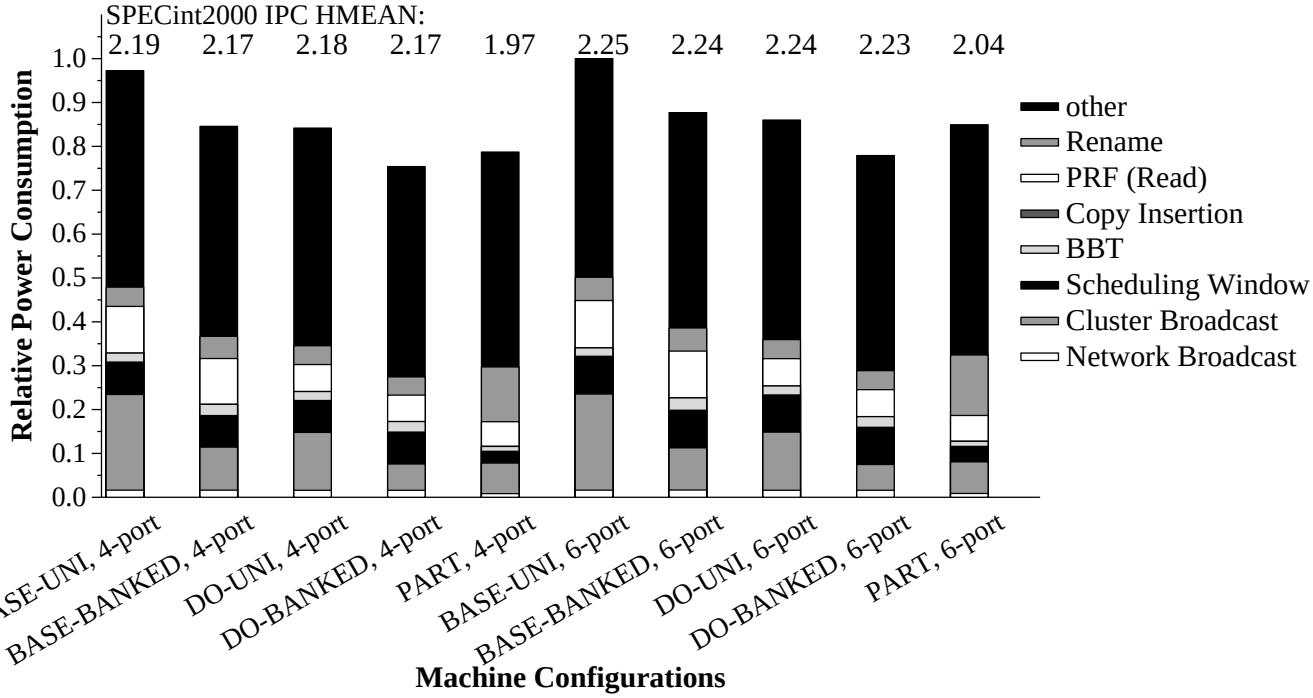


Figure 4. Relative Power Consumption

Figure 5 shows the IPC of each benchmark for each of the five models using 4 issue ports per cluster. On average, the Baseline processor with a monolithic register file has an IPC less than 1% higher than the model with a banked register file, and the processor with Demand-Only Broadcast has an IPC within 1% of the Baseline with a banked register file. The IPC of DO-BANKED is, on average, 10% higher than the IPC of PART, despite the fact that PART has more scheduling window and register file entries. The discrepancy is due to the fact that the copy instructions increase the length of the data dependence chains. The only benchmark that overcomes this limitation and clearly benefits from having a unified physical register file is `gap`. The power consumption for each individual benchmark, relative to the average power consumption of the Baseline, is shown in Figure 6.

6.1. Effect of Copy Instructions

Copy instructions lower the IPC of the DO-BANKED and PART models. In the PART model, every cluster in which an instruction's result is consumed (other than its own) requires a copy. Figure 7 shows the number of clusters in which a result is consumed. The first bar for each benchmark represents the DO-BANKED model with

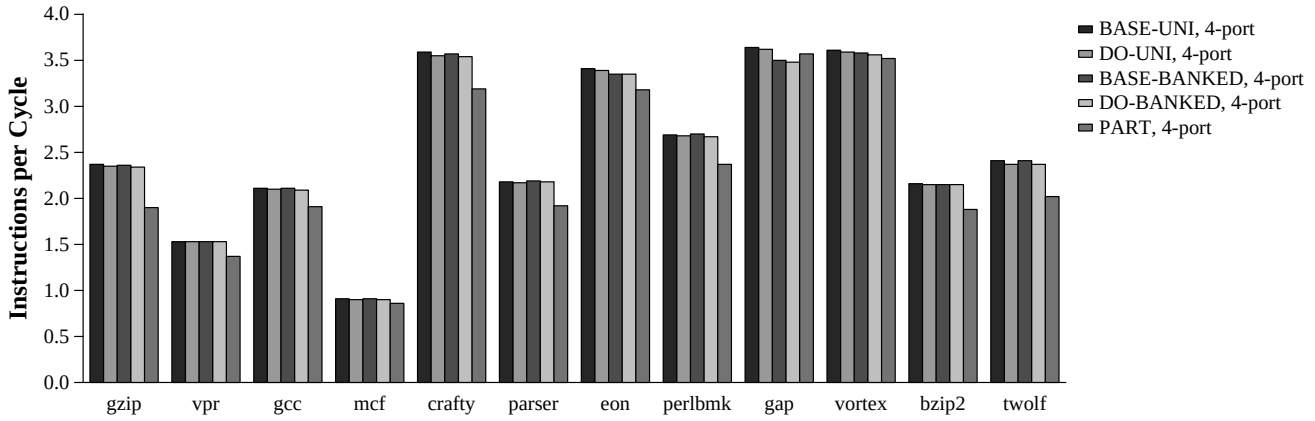


Figure 5. IPC on the SPECint2000 Benchmarks

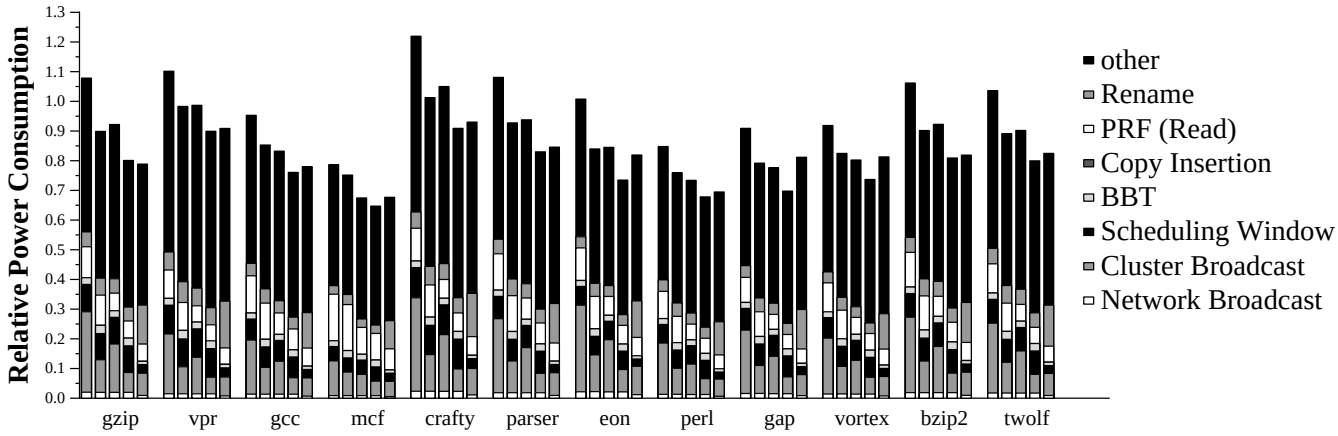


Figure 6. Power consumption relative to BASE-UNI average. Models (4 issue ports), from left to right: 1) BASE-UNI, 2) BASE-BANKED, 3) DO-UNI, 4) DO-BANKED, 5) PART

4 issue ports per cluster; the second bar is the PART model. The third and four bars are the DO-BANKED and PART models, respectively, with six ports. Half the time, a result is only needed in its own cluster. On average, with the Demand-Only technique, there are 1.6 register file writes per architected register destination, compared to 4 in the Baseline model. Note that this metric does not represent the number of copy instructions for the Demand-Only model because copy instructions are not always needed when a consumer resides in a different cluster from its producer. The percentages at the top of each pair of bars represent the fraction of copy instructions that are needed in the DO-BANKED model relative to the PART model for the 4-port and 6-port configurations. For example, in `gzip`, the DO-BANKED, 4-port model requires 6% as many copy instructions as the PART 4-port model.

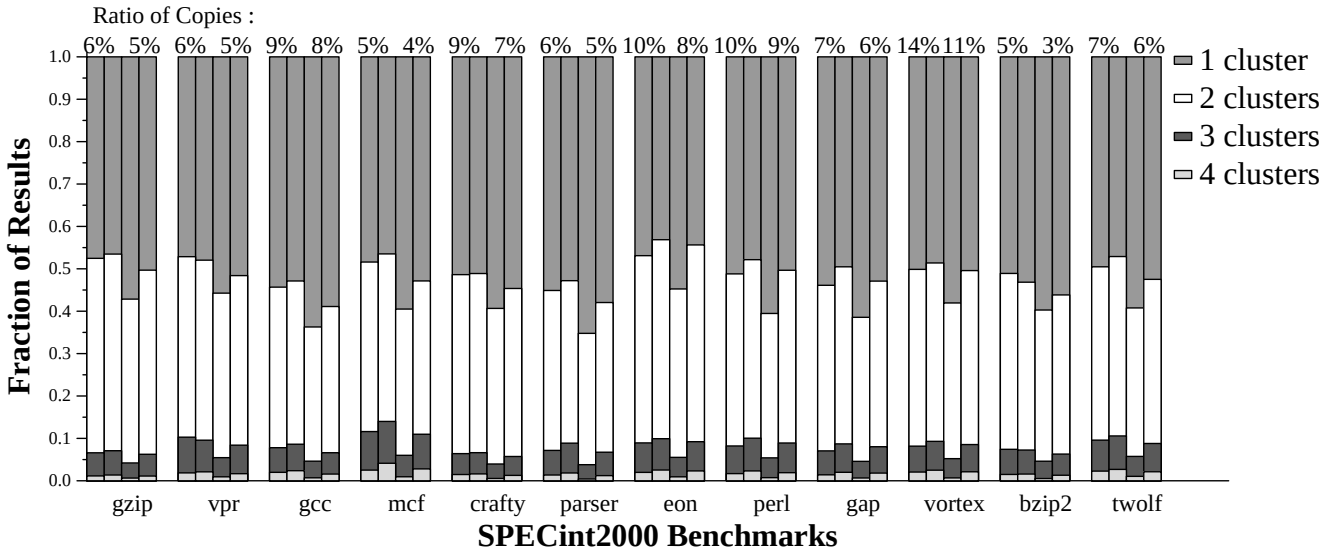


Figure 7. Number of clusters in which result is used. Models: 1) DO-BANKED, 4-port, 2) PART, 4-port, 3) DO-BANKED, 6-port, 4) PART, 6-port

6.2. Scheduling Window Write Ports

There are many instructions that cannot be placed in the same cluster as their source operand because there are no available write ports to the desired scheduling window. Additional write ports will make the scheduler slower, but it will increase the chance that instructions can be placed near their source operands. This is apparent from comparing the 4-port and 6-port configurations in Figure 7. Figure 8 shows the IPC for all benchmarks using the BASE-BANKED, DO-BANKED, and PART models, using 4 or 6 write ports per cluster. On average, the IPC improves by 2.9%, 3.2%, and 3.3%, respectively, when increasng from 4 to 6 ports per cluster.

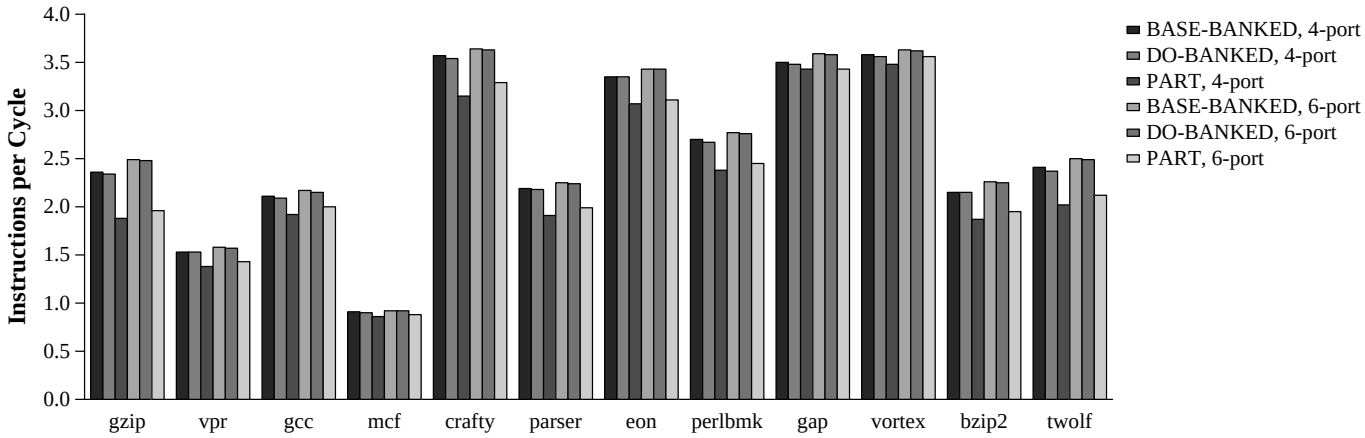


Figure 8. Effect of the number of issue ports on IPC.

Figure 9 shows the percentage of instructions with at least one source operand that are placed in their desired cluster: the cluster containing their first source operand. Note that most instructions get their top steering preference in the models with the fewest copy instructions (or no copy instructions, in the case of BASE-BANKED), since copy instructions occupy issue ports. This demonstrates that copy instructions can have a negative-feedback effect: because copy instructions occupy scheduling window write ports, they cause other instructions to be steered to an undesired cluster, thus creating even more copy instructions.

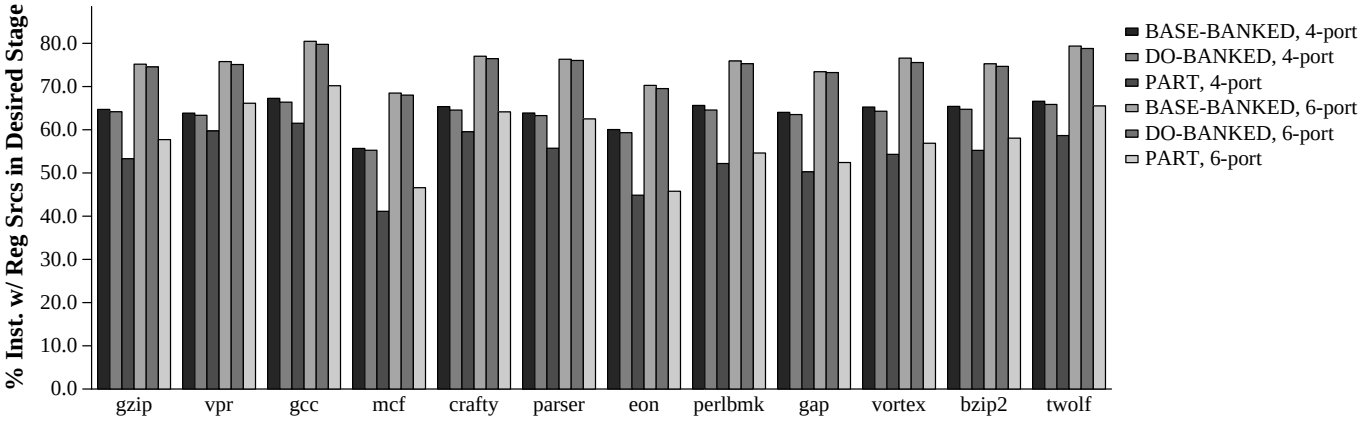


Figure 9. Effect of the number of issue ports on steering.

7. Conclusion

This paper has demonstrated that the physical register file is a large source of power consumption in clustered processors, and Demand-Only Broadcast is an effective technique for reducing this power. This technique was evaluated in a 16-wide clustered processor, although it is applicable in clustered processors with narrower issue widths as well. In a processor with 4 clusters, it reduces the number of register writes from 4 to 1.6 per register-updating instruction. It reduces total processor power consumption of a high-performance clustered processor by 10% while impacting IPC by less than 1%.

8 Acknowledgements

We would like to thank members of the HPS research group, Antonio González, and anonymous reviewers for their comments and insights on previous drafts of this paper. This work was supported in part by donations from Intel, an IBM Ph.D. Fellowship, and a UT College of Engineering Doctoral Fellowship.

References

[1] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 237–248, Dec. 2001.

[2] A. Baniasadi and A. Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. In *Proceedings of the 33th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 337–347, Dec. 2000.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[4] M. Butler and Y. Patt. An investigation of the performance of various dynamic scheduling techniques. In *Proceedings of the 25th Annual ACM/IEEE International Symposium on Microarchitecture*, 1992.

[5] R. Canal, J.-M. Parcerisa, and A. Gonz´alez. Dynamic cluster assignment mechanisms. In *Proceedings of the Sixth IEEE International Symposium on High Performance Computer Architecture*, Feb. 2000.

[6] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 149–159, Dec. 1997.

[7] K. I. Farkas, N. P. Jouppi, and P. Chow. Register file design considerations in dynamically scheduled processors. In *Proceedings of the Fourth IEEE International Symposium on High Performance Computer Architecture*, pages 40–51, 1998.

[8] B. A. Gieseke, R. L. Allmon, D. W. Bailey, B. J. Benschneider, S. M. Britton, J. D. Clouser, H. R. F. III, J. A. Farrell, M. K. Gowan, C. L. Houghton, J. B. Keller, T. H. Lee, D. L. Leibholz, S. C. Lowell, M. D. Matson, R. J. Matthew, V. Peng, M. D. Quinn, D. A. Priore, M. J. Smith, and K. E. Wilcox. A 600MHz superscalar RISC microprocessor with out-of-order execution. In *1997 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 176–178, Feb. 1997.

[9] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the intel pentium 4 processor. *Intel Technology Journal*, Q1, 2001.

[10] G. A. Kemp and M. Franklin. PEWs: A decentralized dynamic scheduler for ILP processing. In *Int. Conference on Parallel Processing*, pages 239–246, 1996.

[11] S. Palacharla, N. P. Jouppi, and J. E. Smith. Quantifying the complexity of superscalar processors. Technical Report TR-96-1328, Computer Sciences Department, University of Wisconsin - Madison, Nov. 1996.

[12] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[13] J.-M. Parcerisa and A. Gonz´alez. Reducing wire delay penalty through value prediction. In *Proceedings of the 33th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 317–326, 2000.

[14] J.-M. Parcerisa, J. Sahuquillo, A. Gonz´alez, and J. Duato. Efficient interconnects for clustered microarchitectures. In *Proceedings of the 2002 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, 2002.

[15] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *The Journal of Supercomputing*, 7:9–50, 1993.

[16] A. Seznec, E. Toullec, and O. Rochecouste. Register write specialization register read specialization: A path to complexity-effective wide-issue superscalar processors. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 383–394, 2002.

[17] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, 1995.

[18] J. Stark, M. D. Brown, and Y. N. Patt. On pipelining dynamic instruction scheduling logic. In *Proceedings of the 33th Annual ACM/IEEE International Symposium on Microarchitecture*, 2000.

[19] K. C. Yeager. The MIPS R10000 superscalar microprocessor. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 28–41, 1996.

[20] V. V. Zyuban and P. M. Kogge. The energy complexity of register files. In *Proceedings of the 1998 International Symposium on Low Power Electronic Design*, pages 305–310, 1998.

[21] V. V. Zyuban and P. M. Kogge. Inherently low-power high-performance superscalar architectures. *IEEE Transactions on Computers*, 50(3):268–286, Mar. 2001.