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

This paper introduces a technique called Demand-Only Broadcast that reduces the power consumption of the register file and result bypass network in a clustered execution core. With this technique, an instruction's result is only broadcast within 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 intra-cluster broadcasts, the total processor power consumption (including the hardware needed by this mechanism) is reduced by 10%, while having less than a 1% impact on IPC. Demand-Only Broadcast also results in a 10% higher IPC and 4% lower power consumption than a clustered processor with a partitioned register file.

1. Introduction

Many high-performance processors use large instruction windows to exploit ILP. Instruction windows may be partitioned into *clusters* to reduce scheduling latency and the minimum data forwarding delays. By steering dependent instructions to the same cluster, most of the inter-cluster forwarding 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 wide-issue processors, such as the Alpha 21264 [6], duplicate 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 of each copy, and thus the access latency, is 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 within the clusters that contain its consumers at the time that its scheduling tag is broadcast. If a consumer is fetched and issued to a remote cluster *after* the producer's tag was broadcast and the producer's result was not written to the remote cluster, then a *copy instruction* must be inserted to broadcast the result in the remote 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.

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 intra-cluster tag broadcasts by 59%. While both of these models have the same register file latency, the total power consumption is reduced by 10% while having less than a 1% impact on IPC. The Demand-Only model is also compared 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 8% higher IPC and 4% lower power consumption than the processor with a partitioned register file.

Section 2 discusses related clustering techniques which limit result broadcast. Section 3 explains the baseline processor used to evaluate Demand-Only Broadcast, and Section 4 describes its implementation. Section 5 describes a previously published model to which we compare our results. Sections 6 and 7 explain the experimental framework and the results, and Section 8 concludes.

2. Related Work

There are several clustering paradigms that limit cluster communication. This paper will discuss those microarchitectures which implement sequential ISAs [14].

In the Multiscalar processing paradigm [16], a program's instruction stream is divided into contiguous sections called tasks which are executed concurrently on several processing units. Because there may be data dependences between the tasks, the live-out 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-outs, and inserts instructions called *release instructions* into the code to indicate which values should be forwarded.

Several clustered processors [8, 9, 20] use a centralized instruction fetch unit but steer instructions to one of several execution clusters. All of these paradigms rely on buffers or similar mechanisms to forward data between clusters. These buffers increase the latency for forwarding values between clusters because the copy or forwarding operations must be scheduled. Demand-Only Broadcast, however, does not use forwarding buffers.

In the Multicluster Architecture [4], 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 needs a register operand from another cluster, copies of the instruction must be inserted into more than one cluster. These extra instructions must contend with regular instructions for scheduling window 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.

The architecture described by Canal, Parcerisa, and González [3, 12] also partitions the physical register file, scheduling window, and 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 *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. This paradigm is compared against Demand-Only Broadcast, and it is explained in more detail in Section 5.

3. Baseline 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. Figure 1 shows an overview of the execution core. Each cluster holds one fourth of the scheduling window entries, and like the Alpha 21264 [6], each cluster contains a copy of the physical register file. The pipeline is shown in Figure 2. 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. Instructions are fetched and decoded in the first 4 cycles. In the next 5 cycles, instructions are renamed and steered to a cluster. After instructions are assigned to a particular cluster, they are issued (i.e. inserted into the cluster's scheduling window). After an instruction becomes ready and is selected for execution, it reads the register file and then executes.

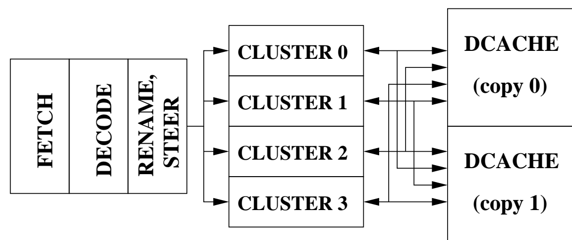

Figure 1. Execution Core

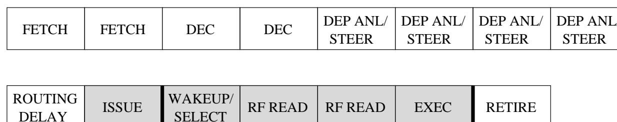

Figure 2. pipeline

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 load access la-

tency. Stores must write data to both copies, but loads read from only the closest cache.

Figure 3 shows the contents of one cluster. Each cluster contains a copy of the Busy-Bit Table [18], the local scheduling window, 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 used in processors with special-purpose functional units.

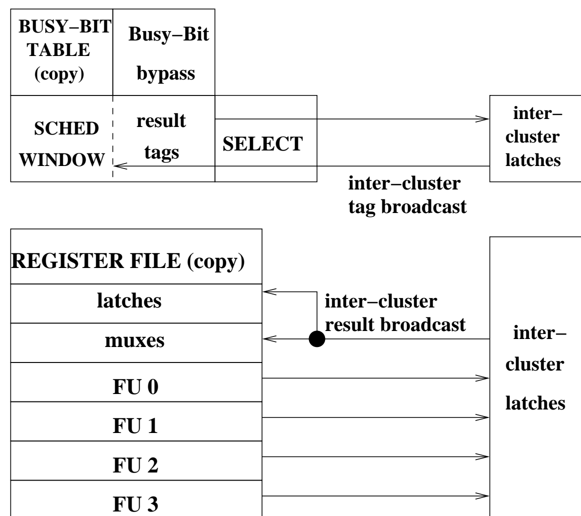

Figure 3. Logic within one Cluster.

3.1. Instruction Steering

The performance of a clustered processor is sensitive to the steering mechanism used [1, 12]. 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 [1, 3], dependence-based [11], predicted Last-Source-Ready [17], and the DCOUNT threshold [12].

A dependence-based steering algorithm was used in the experiments for this paper. 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.[1] If it has no source operands, its cluster preference is assigned according to a Modulo-4 heuristic. 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 assigned to the closest cluster. 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 6.

3.2. Instruction Issue

When an instruction is first placed into the scheduling window, the Busy-Bit Table (BBT) is read to determine if its source operands are ready. The BBT, 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 a source operand's BBT entry is set or its tag is broadcast in that cycle, then the instruction sets the Ready bit in its reservation station entry. If the BBT entry is clear and the tag is not broadcast in that cycle, then the Ready bit is not set. A BBT entry is cleared when a new instruction is assigned to the corresponding physical register.

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 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.

3.3. Instruction Scheduling

The scheduling window holds instructions that are waiting to execute. Each entry holds the physical register numbers and Ready bits for an instruction's source operands. 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. Since an instruction's consumers may reside in any cluster, it broadcasts its destination tag and result to all clusters. Because instructions are scheduled for execution several cycles before they execute, their tags are broadcast several cycles before their data is broadcast.

[1] 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 had been produced would further complicate the steering logic, and it may require adding additional ports to the BBTs.

3.4. The Register File

The register file used in the baseline model is banked by using Register Write Specialization, first described by Seznec et al [15]. The register file is divided into four banks in order to reduce the number of write wordlines per bit cell. All instructions in the same 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 the unified register file. All instructions must still broadcast 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. However, simulations comparing it to an unbanked register file showed that this affected IPC by less than 1%. The register file described here is replicated across all four clusters to reduce the number of read ports.

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. The Busy-Bit Table in each cluster keeps track of which physical registers are needed by instructions in the cluster. Rather than just 1 bit for each BBT entry, there are two: the "Broadcast bit" indicates if the tag has been broadcast to that cluster, and the "Use bit" indicates if there are any instructions within that cluster requiring that physical register. BBT entries are reset when an architectural register is first mapped to a physical register, just as in the Baseline.

When an instruction is first placed into a cluster, it reads the BBT entries corresponding to its source operands. If both the Use and Broadcast bits of an entry are set, then the source operand is available and the corresponding Ready bit in the instruction's reservation station entry is set. The case where the Broadcast bit is set but the Use bit is clear is discussed in Section 4.1. The instruction sets the Use bits of those BBT entries, if they are not already set, as well as the Use bit for the BBT entry of its own destination physical register number.

When an instruction's tag is broadcast to a cluster, the Broadcast bit for its destination register is set, just as in the baseline. 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 *intra-cluster data broadcast* (i.e. the broadcast within the local cluster) will be blocked. This may add one gate to the data path, depending on the implementation.

There is plenty of time to set the controls to gate the intra-cluster data broadcast and prevent the register file write. This is because 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 last scheduling stage and the last execution stage for a majority of integer instructions). When the Use bit is read, it will enable the latch for the data result bus N cycles later. N is generally at least as large as the minimum number of cycles for the register file access plus execution, and will increase as pipeline depths increase. For example, N is 5 cycles in the Intel Pentium 4 [7].

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 BBT entry in Cluster 0 corresponding to A's destination register, and BBT-3[A] refers to the BBT entry for A 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 broadcasted.

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].bc = 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].bc = 1. Since BBT-3[A].use is 1, don't block data broadcast in cycle 5. *B* wakes up.
4	*B* is selected for execution and broadcasts its tag.
5	*A*'s data is broadcast to Cluster 3.

Table 1. Timing for an inter-cluster broadcast.

4.1. Copy Instructions

In the previous example, if instruction B were issued to cluster 3 after A's tag was broadcast to that cluster and the Use bit for A's BBT entry in cluster 3 (BBT-3[A].use) was not set, then A's data would not be broadcast to that cluster in cycle 5. In this situation, a *copy instruction* will be required to re-broadcast the result. The copy 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). After being scheduled, it 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 contents 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. The instruction's Ready bit is not set.

Insertion of Copy Instructions

In each cluster there is 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 this 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 bit of the vector corresponding to that physical register 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 process of updating this vector is pipelined over two cycles to account for wire delays, and it 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 there is a 2-cycle delay between the clusters' issue logic and the steering logic, which accounts for the delay for updating the Copy Request Vector. Performance is relatively insensitive to this delay since the scenario where copy instructions are needed is rare.

The example in Table 2 illustrates the scenario where instruction A's tag is broadcast before B is issued. Initially, *A* is in Cluster 0 and *B* will later be issued to Cluster 3. When *A*'s tag is broadcast 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 in cycle 4, 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. In cycle 5, A's data broadcast is blocked even though B has been issued because the control signals have already been set. By cycle 6, the bit of the Copy Request Vector corresponding to A's destination register has been set and the steering logic inserts a copy instruction. In cycle 9, the copy instruction is issued into Cluster 0, and its tag is broadcast to Cluster 3 in cycle 12. B's execution was delayed by 9 cycles because it missed the tag broadcast.

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. Set BBT-0[A].bc = 1.
3	*A*'s tag broadcast to Cluster 3. Read BBT-3[A].use and set BBT-3[A].bc = 1. Block data broadcast (2 cycles later).
4	*B* is issued to Cluster 3. BBT-3[A].use = 1 and BBT-3[A].bc = 0. Request copy.
6	CRV[A] is set.
9	*Copy-A* is issued (already awake) and selected.
12	*Copy-A* broadcasts tag in Cluster 3; *B* wakes up. Set BBT-3[A].bc.
13	*B* is selected and broadcasts its tag to Cluster 3.

Table 2. Timing for an Intra-Cluster Broadcast Requiring a Copy Instruction.

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 7 will show power and performance results.

5. Partitioned Register File Model

We compared Demand-Only Broadcast to another machine model that reduces register file and broadcast power by using a partitioned register file. This model 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. This model is similar to the paradigm used by Parcerisa and González [12].

Copy instructions must be used to forward data between clusters. 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. [13], our simulations showed that adding more bypass buses and ports did not significantly help IPC. However, we note that 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 a performance bias towards the Demand-Only model, we will assume that this model can issue a copy instruction instantaneously instead of taking five cycles as in the Demand-Only model. 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 registers 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. [13] (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 windows in this model are 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 comparison of the power consumption.

6. Experimental Framework

We have measured the IPC and per-cycle power consumption for three processor models: the baseline processor (BASE), the baseline using Demand-Only Broadcast (DOB), and the model with the partitioned register file (PART). Our simulator is a cycle-accurate, execution-driven processor which models mispredicted-path effects and executes the Alpha ISA[2]. Our power model is based on the Wattch framework [2]. Wattch models switching power given the amount of switching activitiy in individual components on the chip. It has

[2]In response to a reviewer's comment, we note that we are not using any of the frequently used publically available performance simulators. Our simulator was written from scratch by members of our research group.

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 measuring the switching activity factor and the maximum power consumption of individual components have been modified. This section discusses some of 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 in our model. The most obvious 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 our register files, writes also have a disproportionate power dissipation compared to reads [19]. 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 [5]. 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 3 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 from and written to 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 [10] 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 the original Wattch model 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, and logic for inserting copy instructions. The processor configurations

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 Exec 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
Unifi ed 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

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

Table 4. Model-Specific 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* 19% of their maximum every cycle, even when they are not accessed *or* fewer than 19% of the ports are accessed.

7. Results

We have evaluated the three models on the SPECint2000 benchmarks. Smaller input sets were used on some benchmarks to reduce simulation time. The average power estimates shown in this section are based on "per-cycle" power estimates of all processor components, although not all configurations may run with the same cycle time. The per-cycle power dissipation for each model, relative to the Baseline average, is shown in Fig-

ure 4. Each bar shows the contribution of each of the processor units to the total processor power consumption. The components are shown in the order listed in the graph's legend. 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. These include components of the execution core as well as instruction, data, and level 2 caches; translation tables; and load-store buffers. The bottom two categories measure all dynamic power consumption due to result broadcasts and register file writes. **Inter-Cluster Broadcast** is the power consumption of the horizontal data buses running between the clusters. In the DOB model, *all* results are broadcast across the entire inter-cluster bypass, although the PART model only broadcasts the results of copy instructions across the network. **Intra-Cluster Broadcast**, is the power for broadcasting a result within one or more clusters, and includes the power consumption for register file writes. The power dissipation in this category is dominated by the register file write, not the bypass buses. This is the component of power that is directly affected by using Demand-Only Broadcast. The **PRF Read** component includes all dynamic power consumption for the register file reads, in addition to the power 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, and is not visible on the graph. 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 DOB model because the rename logic had to keep track of four mappings per architectural register. It benefits from having fewer scheduling tag broadcasts, though. The power consumption for the DOB model is 10% lower than that of the BASE model, and 4% lower than that of the PART model.

Figure 5 shows the IPC of each benchmark for each model. On average, the DOB model has an IPC within 1% of BASE and 10% higher than the PART model, despite the fact that PART has more scheduling window and register file entries. The PART model's IPC is lower because copy instructions increase the length of the data dependence chains.

Table 5 shows an average distribution of the number of clusters in which register values are consumed. There was little variance among benchmarks. 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. Even though values are needed in only slightly more clusters in the PART model, it requires 16 times as many copy instructions as the DOB model because every cluster in which an instruction's result is consumed (other than the producing cluster) requires a copy. Furthermore, copy instructions can have a negative-feedback effect: because copy instructions occupy resources, they may cause other instructions to be steered to an undesired cluster, thus creating even more copy instructions.

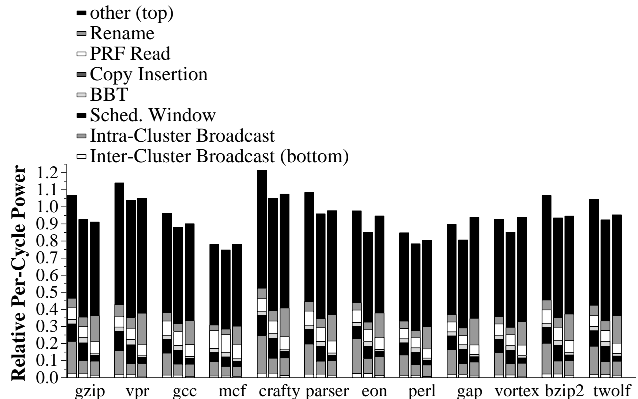

Figure 4. Relative Power consumption. BASE, DOB, PART models shown left to right.

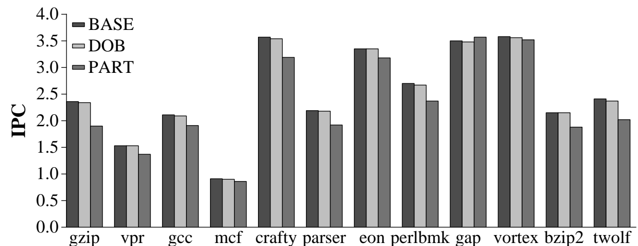

Figure 5. IPC on SPECint2000 Benchmarks

Num clusters	1	2	3	4
DOB	50.5%	41.2%	6.5%	1.7%
PART	48.9%	41.8%	7.0%	2.3%

Table 5. Fraction of results with given number of cluster broadcasts.

8. 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%.

9 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] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] B. A. Gieseke et al. 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.

[7] 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.

[8] M. S. Hrishikesh. Design of wide-issue high frequency processors in wire-delay dominated technologies. 2004.

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

[10] 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.

[11] 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.

[12] 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.

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

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

[15] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 295–306, 2002.

[16] 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.

[17] 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.

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

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

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