Microarchitectural Trade-offs in the Design of a Scalable Clustered Microprocessor

Abstract

Clustered microarchitectures are an attractive alternative to large monolithic superscalar designs due to their potential for higher clock rates in the face of increasingly wire-delay-constrained process technologies. In such a microarchitecture, the distribution of functional units, the register files, and the issue queues across multiple clusters reduces the latency of various cycle time critical paths, thereby enabling a faster clock. However, a penalty in terms of instructions per cycle is incurred if instructions frequently communicate values among clusters because of dependences.

In this paper, we propose several novel extensions that significantly improve the performance of clustered designs. First, we explore a word-interleaved clustered cache in which memory instructions are steered to clusters based on addresses, and when the effective address is unknown, directs memory operations to the appropriate cluster via bank prediction. We then study the scalability of the resulting clustered microarchitecture as the number of clusters is increased (resulting in a corresponding increase in inter-cluster communication latency). Our evaluation identifies the key bottlenecks and shows how novel enhancements to the cluster resource allocation mechanisms can significantly improve the scalability of the design. We also show that communication latency in a highly clustered processor can be reduced for certain programs by only using a subset of the clusters. Overall, these enhancements achieve a 30% improvement over a baseline design with the clustered cache.

1 Introduction

The extraction of large amounts of instruction level parallelism (ILP) from common applications in modern processors requires the use of many functional units and large on-chip structures such as issue queues, register files, caches, and branch predictors. As CMOS process technologies continue to shrink, wire delays become dominant (compared to logic delays) [19, 21]. This, combined with the continuing trend toward faster clock speeds, increases the time in cycles to access regular on-chip structures (caches, register files, etc.). Not only does this degrade instructions per cycle (IPC) performance, it also presents various design problems in breaking up the access into multiple pipeline stages. In spite of the growing numbers of transistors available to architects, it is becoming increasingly difficult to design large monolithic structures that aid ILP extraction without increasing design complexity, compromising clock speed, and limiting scalability in future process technologies.

As a solution to this problem, a number of recent proposals [1, 5, 7, 8, 13] distribute the key processor resources across multiple *clusters*, each of which contains a subset of the issue queues, register files, and the functional units. In such a microarchitecture, at the time of instruction rename/dispatch, each instruction is steered into one of the clusters. As a result of decreasing the size and bandwidth requirements of the issue queues and register files, the access times of these critical cycle time structures are greatly reduced, thereby permitting a faster clock. An additional advantage is reduced design complexity due to the simplification of these structures and the fact that the homogeneity of the clusters allows a single cluster to be replicated.

However, the primary disadvantage of clustered microarchitectures is their reduced IPC compared to a monolithic design. Although single-cycle intra-cluster bypassing can be achieved, inter-cluster bypassing requires that the value be copied into the dependent instruction's cluster register file, thus incurring an additional latency. An additional constraint is the reduced instruction window and issue width in each cluster may degrade overall issue width relative to a single, large window, wider issue machine. Breaking the processor resources into more and more clusters is a feasible option only if the associated IPC degradation is not large enough that it offsets the effect of the improved clock speed.

One of the attractive features of a clustered microarchitecture is the reduced design effort in producing successive generations of a processor. Once a single cluster core has been designed, more of these cores can be put into the processor for a very low design cost as the transistor budget increases. Hence, the clustered microarchitecture is very scalable in terms of design complexity. Further, in a simultaneous multithreaded processor [27], the greater the number of clusters (and instruction fetch units), the better the instruction throughput. Both these factors make a highly-clustered processor in the billion transistor era an extremely attractive option. However, the scalability of single thread performance with increasing number of clusters is not guaranteed as the increased communication costs could offset the benefit of providing more resources.

In this paper, we explore the scalability of the clustered microarchitecture as the number of clusters is increased and as the number of resources in each cluster is reduced. As processors become increasingly communication bound, the use of centralized structures can degrade performance. Most studies on clustering assume centralized instruction fetch and dispatch stages and a centralized data cache. We show that the former assumption is not a great impediment to performance. However, a centralized data cache that may be many cycles away from the most distant cluster could be a serious limiter of performance. We show that a clustered cache, where each cache bank is associated with a certain cluster, and a bank predictor to direct memory operations to the appropriate cluster, can greatly reduce the average load latency.

With this improved base case, we show that as the size of the cluster core decreases, instruction dispatch becomes a serious bottleneck. This happens because of the increased register replication and the fewer effective registers in each cluster. We propose a number of mechanisms that help alleviate this bottleneck. The first of these directs instructions to neighboring clusters if the assigned cluster is full, thereby increasing the communication costs, but enabling larger in-flight windows. The second mechanism eagerly frees up replicated registers without waiting for the in-order commit process. We also show that intentionally stalling the dispatch stage for a period improves the effectiveness of the heuristic that steers instructions to the clusters. Finally, we show that performance does not always scale up as the number of clusters increases. In a conventional processor, in almost all cases, as the number of resources and the size of the in-flight instruction window are increased, IPC performance also improves. But in a clustered processor, when more clusters are added, the benefit of a larger window can be negated by the increased cost of communication. We show that an organization that has many clusters, but chooses to use fewer clusters for certain applications, helps reduce this communication cost, thereby greatly improving performance.

The rest of the paper is organized as follows. Section 2 begins by describing the simulation infrastructure we used to generate the results we describe in the rest of the paper. Section 3 describes the base clustered microarchitecture. In Section 4, we evaluate the use of the clustered cache, which we then use as our baseline. In Section 5, we explore the scalability aspects as the number of clusters is increased and as the resources in each cluster are reduced. Section 6 evaluates mechanisms that help alleviate the main bottlenecks in the design. We describe related work in Section 7 and conclude in Section 8.

2 Simulation Methodology

Since our evaluation is presented along with the description of our design, we begin by describing the simulation infrastructure and benchmarks we used for the evaluation. Our simulator is based on Simplescalar-3.0 [6] for the Alpha AXP instruction set. The simulator has been modified to represent a microarchitecture resembling the

Fetch queue size	64
Branch predictor	comb. of bimodal and 2-level; bimodal size 2048; Level1 1024 entries, history 10; Level2 4096 entries Combining predictor size 1024; RAS size 32; BTB 2048 sets, 2-way
Branch mispredict penalty	12 cycles
Fetch width	16 (across up to four basic blocks)
Dispatch, issue, and commit width	16
ROB and Ld/St queue	400 and 200
Issue queue size	160 (int and fp, each)
Register file size	400 (int and fp, each)
L1 D-cache	128KB 2-way
L1 I-cache	64KB 2-way
L2 unified cache	1.5MB 6-way, 20 cycles
TLB	128 entries, 8KB page size (I and D, each)
Memory latency	100 cycles for the first chunk
Memory ports	16 (interleaved)
Integer ALUs/mult-div; FP ALUs/mult-div	16/16; 16/16

Table 1. Simplescalar simulator parameters for the base 16-way processor.

Alpha 21264 [16]. The register update unit (RUU) is decomposed into issue queues, physical register files, and the reorder buffer (ROB). The issue queue and the physical register file are further split into integer and floating-point. The memory hierarchy is also modeled in detail (including interleaved access, bus and port contention, writeback buffers, etc). This base processor model was modified to model the clustered microarchitecture. The functional units, registers, and issue queue entries were distributed equally among the various clusters. We simulated an aggressive 16-way processor for most of our experiments. While we did not model a trace cache, we assumed that instructions could be fetched from up to four basic blocks at a time. The important simulation parameters are summarized in Table 1.

As a benchmark set, we used a variety of programs from SPEC95, SPEC2000, the Olden suite [24], and the UCLA Mediabench [17]. This collection of programs was chosen to represent a mix of various program types - integer, floating-point, memory intensive, branch predictor limited, etc. The details on these programs are listed in Table 2. The programs were compiled with Compaq's cc, f77, and f90 compilers for the Alpha 21164 at the highest optimization level. We studied cache miss rate traces to identify warm-up phases for the programs and fast-forwarded past these instructions. Another one million instructions were simulated in detail to prime all structures before doing the performance measurements over the next 10M instructions. The vast number of simulations and the slowness of the simulator prevented us from simulating longer instruction windows, which are possibly more representative of overall program behavior. However, the simulation is long enough that most processor structures (caches, predictors, etc) are warmed up and our choice of benchmark programs ensures that various types of code patterns are simulated. We selectively verified this via longer runs of up to a 100 million instructions for certain configurations, which produced essentially the same results. We will use a complete set of longer runs in the final version of this paper.

3 The Base Clustered Processor

The baseline 4-clustered microarchitecture [1, 5, 7, 8, 13] is shown in Figure 1(a). The branch predictor and instruction cache are centralized structures, just as in a conventional processor. Once instructions have been brought into the instruction fetch queue (IFQ), they go through the register renaming process. At this point, each instruction also gets assigned to a specific cluster. Each cluster has its own issue queue, register file, a set of functional units, and its own local bypass network. The instruction being renamed is assigned a destination physical register from its own cluster (for example, *cluster A*). If one of its operands is being produced in a

Benchmark	Input dataset	Instr fastfwded	L1 miss rate	Branch prediction accuracy	Baseline IPC
art (SPEC2K FP)	ref	300M	15.5%	95%	1.14
cjpeg (Mediabench)	testimg	200M	0.1%	89%	1.64
djpeg (Mediabench)	testimg	150M	0.0 %	94%	5.79
em3d (Olden)	nodes 1000, arity 10	20M	3.8%	92%	4.55
gcc (SPEC95 Int)	ref	300M	0.6%	91%	1.61
mgrid (SPEC95 FP)	ref	500M	1.0%	98%	4.19
parser (SPEC2K Int)	ref	2000M	1.0%	92%	1.49
su2cor (SPEC95 FP)	ref	1000M	0.0%	99%	2.72
swim (SPEC2K FP)	ref	1000M	3.8%	100%	2.69

Table 2. Benchmark description. Baseline IPC is for a monolithic 16-way processor.

different cluster, *cluster B*, a physical register for that operand is also created in *cluster A*. As soon as this operand is produced in *cluster B*, it is copied into its corresponding physical register in *cluster A* as well. The operand is marked as ready and the instruction can now issue. At the time of instruction commit, all copies of the physical register being freed (the earlier mapping for the logical destination register) are released. Because of the cost of communicating results across clusters, the execution of an instruction might be delayed by a few cycles. Operands that are produced within the same cluster are bypassed to the consuming instruction in a single cycle.

While such designs afford a faster clock, the cost is a noticeable IPC degradation. The three main causes for this are the penalty arising from inter-cluster communication, the stalls caused by load imbalance, and additional dispatch stalls. Note that dispatch is stalled as soon as a cluster is full – it is often better to wait than send an instruction elsewhere and incur a communication cost. Minimizing the overall IPC degradation has been the focus of recent studies [1, 5, 7, 8, 9, 13]. A simple steering heuristic [7] that has been shown to be quite effective steers an instruction to the cluster that produces most of its operands. In the event of a tie or under circumstances where a distinct imbalance in issue queue occupancy is seen, instructions are steered to the least loaded cluster. The choice of various thresholds determines the trade-off between these two conflicting forces. Another steering heuristic [5] that performs well assigns three successive instructions to a cluster before assigning the next three to the next cluster. Our steering heuristic is a generalization of some of these heuristics in the literature. For each instruction, it computes the following function for each cluster and then picks the one that has the highest value according to the following:

$$f(cluster) = DEP\_THRESH \times (num\_dependents) - I\_THRESH \times (num\_issueq\_entries) - R\_THRESH \times (num\_regfile\_entries)$$

The three thresholds determine the priority that is given to data dependences, issue queue load, and register pressure. Each simulated clustered organization was optimized by trying various threshold values and picking the set that performed best. The steering heuristic also incorporates a criticality predictor similar to those proposed recently [14, 28]. A simple table indexed by program counter predicts which of the two operands of an instruction is likely to be produced later and the corresponding data dependence is given a higher weight in the equation.

Once loads and stores are ready, they are inserted into a centralized load-store queue (LSQ). From here, stores are sent to the centralized L1 cache when they commit and loads are issued when they are known to not conflict with earlier stores. The LSQ is centralized because a load in any cluster could conflict with an earlier store from any of the other clusters. While the cache design might itself be distributed (for example, it might be a word-interleaved banked cache [23]), its interface to the clusters is still monolithic. A memory operation issued by any cluster could access any part of the cache. Hence, a load issued by a cluster would have to first propagate its address to the LSQ, where it is disambiguated against memory operations from all clusters and then sent to one of the banks.

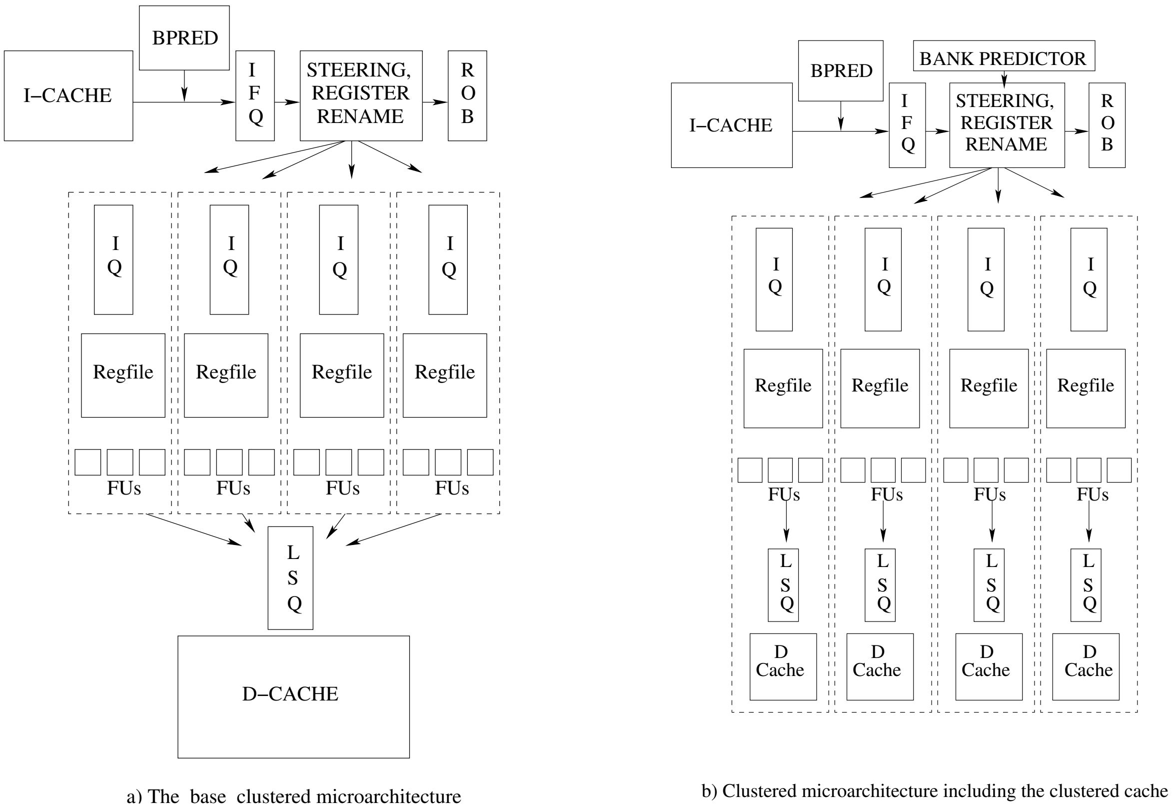

Figure 1. The base clustered microarchitecture with the centralized cache (a) and the clustered cache (b).

4 Clustering the Cache

The use of a centralized LSQ and data cache results in long access times, especially with a large number of clusters. The effective address has to first be propagated to a centralized LSQ, and from there, data might then have to be fetched from the most distant bank. To reduce delays from these long wires, we split the LSQ and the cache into multiple banks and associate each with a different cluster, as shown in Figure 1b. Hence, if there are four clusters, each cluster cache is one-quarter the size of the centralized cache, which significantly reduces access time. Because of the close coupling of each cache bank to a single cluster, the long cross-chip wires of the centralized cache are eliminated.

We impose the restriction that the banks cache mutually exclusive subsets of the virtual address space. This removes the need for cache coherence protocols at the L1 level, thus keeping the design simple and avoiding the use of replicated cache tags. Each cache is serviced by its own LSQ. The goal of the steering mechanism is to steer a load or store to the cluster that caches the corresponding memory address. That way, the instruction naturally progresses from its issue queue, to the corresponding LSQ, and then to its own cache bank. It need not bother with loads and stores in other clusters that are necessarily handling different memory addresses.

4.1 Clustered LSQ Implementation

If an effective address is known when a memory instruction is renamed, then it can be directed to the cluster associated with the appropriate cache bank. However, the effective address is generally not known at rename time, requiring that the bank that this memory operation would access be predicted. Based on this prediction, the instruction is sent to one of the clusters. Once the effective address is computed, it is clear if the prediction was incorrect, in which case, the appropriate recovery action has to be taken.

If the operation is a load, recovery is simple - the effective address is sent to the appropriate cluster, where memory conflicts are resolved in the LSQ, data is fetched from the cache bank, and then returned to the requesting cluster. If the memory operation is a store, the mis-direction could result in correctness problems. A load in a different cluster could have proceeded while being unaware of the existence of a mis-directed store to the same address. Once the mis-direction is discovered, this load and its dependents would then have to be squashed and restarted. To avoid this cumbersome process, we adopt a policy similar to that in [30]. At the time of rename, a store whose effective address is unknown is assigned to a particular cluster (where its effective address is computed), but at the same time, a dummy slot is also created in the other clusters. Subsequent loads behind the dummy slot in other clusters are prevented from proceeding because there is an earlier store with an unresolved address that could potentially cause conflicts. Once the effective address is computed, the dummy slots in all the LSQs except one are removed. This ensures that no recovery has to be done on a store mis-direction. However, this policy does reduce the effective size of the LSQ as some of the entries are replicated until the effective address is computed.

4.2 Bank Prediction

While the organization just described reduces the access time to the cache bank, it can add penalties in the event of a load or store mis-direction. A load mis-direction requires sending the address to the appropriate cluster and the resulting data to be sent back. If the inter-cluster communication cost is a single cycle, the access time for such a mis-directed load increases by two cycles – one to send the address and one to get the data back. Similarly, a mis-directed store that conflicts with a later load would have to communicate its store data to the appropriate cluster before the load can proceed, thus increasing the store-load bypassing cost. Clearly, these cross-cluster communications must be reduced for good performance and it becomes necessary to accurately predict which bank an instruction might access during the steering process. It must also be pointed out that the variable latency of the load (depending on whether it was mis-directed or not) does not significantly complicate the issue stage. Before the load is issued, the effective address is known and hence the latency of the access can be determined. The wake-up operation in the issue queue is accordingly scheduled.

Earlier work by Yoaz et al [29] had looked at the topic of bank prediction, with the intention of using it to improve instruction scheduling. They had also suggested its use in steering memory operations into different memory pipelines. We employ the use of similar bank predictors to determine which bank the instruction will access. The design of these bank predictors is very similar to branch predictors and we use a 2-level predictor for our simulations. At the first level, bank histories are maintained for a number of different instructions (indexed by program counter (PC)). The history is xor-ed with some of the PC bits to index into the second level that maintains 2-bit saturating counters that indicate the predicted bank.

So far, we have assumed that a one-to-one mapping exists between the set of virtual addresses and the set of cache banks. This ensures that the data in each bank is mutually exclusive and mis-directions can be easily detected without any cache coherence protocols. A number of such mappings exist and we start by using the one that is commonly employed in processors [15] - word-interleaving. In a two-bank word interleaved cache, all odd words are placed in one bank and all even words are placed in the other. That way, without the use of true multiporting, in any given cycle, one odd and one even word can be accessed, allowing a read bandwidth of two with a single port. Such a mapping ensures that conflicts for the banks are kept to a minimum, as for most programs, accesses to odd and even words are evenly distributed [23]. This function is easy to generalize to multiple banks and can also be easily computed, given the effective address.

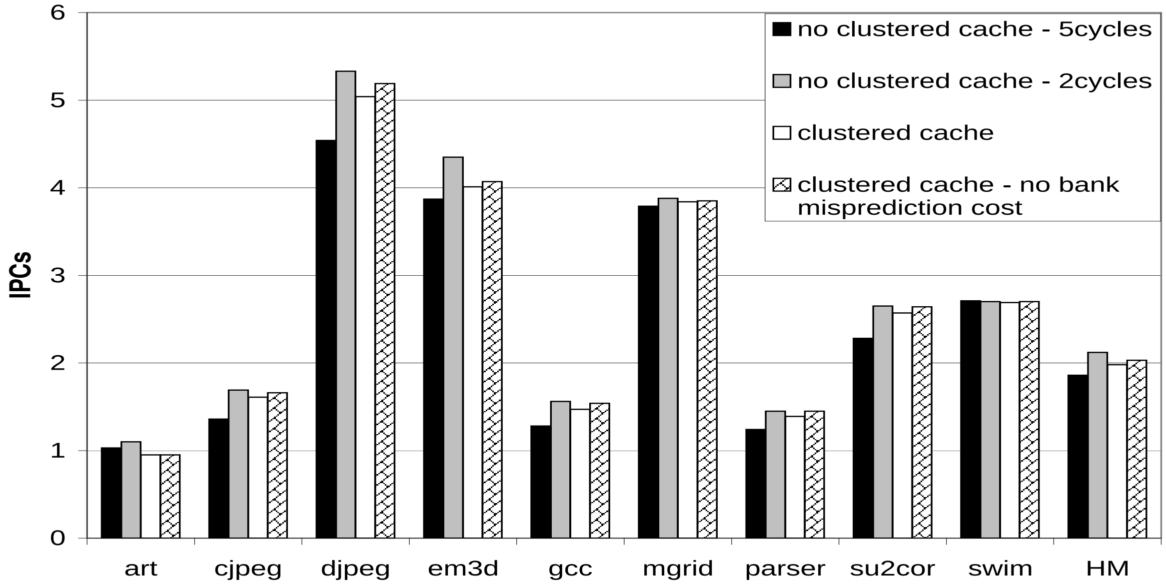

Figure 2. IPCs for a 4-clustered organization with single cycle communication latency. The first two bars show base cases with centralized caches with five and two cycle latencies, the third bar includes the clustered cache and the last bar shows the clustered cache, but with zero bank mispredict penalty.

4.3 Results

We start by showing results for a processor with four clusters. The registers, issue queue entries, and functional units of the processor described in Table 1 are distributed among the four clusters. We assume that the cost of communication between any two clusters is a single cycle. The baseline clustered organization assumes a centralized 128KB 2-way set associative cache. According to the cache access time model of McFarland [20], the access time for such a cache at 0.1μ technology is 2.43ns. We assume that such an access would take three cycles (for a 1.2GHz processor). Since we are assuming a single cycle communication cost in going across the chip, we add two more cycles to the data cache access latency - one for propagating the address to the cache and another for propagating the data back to the clusters.

When the cache is clustered, each cache access only looks up 32KB of data. The access time for such a cache is 1.48ns according to McFarland. We assume that this access takes only two cycles since the cache is closely coupled to each cluster, as opposed to the five cycles for the base case. The two-level bank predictor has 1024 entries at the first level. The instruction PC is used to index into one of these entries, each of which contains a 12-bit history. The history is xor-ed with 12 bits from the PC and this is used to index into the 4096-entry second level, where the contents of the saturating counter indicate the prediction. The steering heuristic not only takes register data dependences and cluster load balance into account, it also considers the recommendation of the bank predictor for loads and stores. The fact that a load could access a certain bank is treated as another data dependence arising from that cluster. We assume a word size of eight bytes for our initial experiments. All words with addresses of the form $4N$ go in the first cluster, of the form $4N + 1$ go in the next cluster, and so on.

Figure 2 shows the effect of the clustered cache on IPC for the various programs. The first two bars show the base clustered processor with data cache access latencies of five and two cycles respectively. The third bar shows IPC with the clustered cache, where each correctly directed load takes two cycles and each mis-directed load takes four cycles. The fourth bar assumes that the mis-directed loads only take two cycles, thus showing the effect of cache clustering on the steering heuristic. The last set of bars represent the overall IPC numbers (represented as harmonic mean (HM) of IPCs). It can be seen that the clustered cache performs about 6% better than a centralized 5-cycle cache because of its reduced average latency. When compared to the 2-cycle centralized cache (an unrealistic model), it performs about 7% worse. This happens for two reasons - (i) the additional latency incurred on a bank misprediction, (ii) the additional register communication because the steering heuristic now tries to also minimize bank mispredictions. The last bar (that assumes zero bank misprediction penalty) shows that more of the IPC degradation seems to come from the latter factor (about 4%). The bank predictor has an average accuracy of 59%. (The accuracy is actually higher, but the instruction is not always steered to the advised bank because the steering heuristic takes register dependences and load into account.) Note that these results assume a single cycle for cross-cluster communication, which is very optimistic for the future. As this cost increases, the access time of the centralized cache goes up greatly and the IPC improvement from using a clustered cache and an accurate bank predictor goes up.

We also tried the effect of different word sizes and found that a word size of eight bytes worked best. Word sizes of 64 and 512 performed about 8 and 13% worse. A larger word size improves the bank prediction rate, but it also affects the distribution of accesses. The greater contention for each cache port as the word size increases has a much bigger impact on performance. For example, in going from a word size of eight to 512, the bank prediction rate improves by 3%, but the number of bank conflicts goes up by a factor of nearly six. We also noticed that a small bank predictor (64-entry L1, 256-entry L2) performs within 1% of the larger one (with a bank prediction accuracy of 50%).

5 Scalability of Performance

For the rest of the paper, we focus on the scalability aspects of the clustered microprocessor assuming the use of a clustered cache. We look at scalability along two axes - (i) Given a large monolithic processor, what is the IPC degradation in splitting it up across multiple clusters? This tells us the amount of clock speed reduction that would be required to make this change worthwhile. (ii) Given a small cluster core, what is the IPC improvement to be had by adding more of these cluster cores? This demonstrates the scalability in performance of a given design as it moves across successive generations.

While looking at such a scalable design, with as many as 16 clusters, we have to make more realistic assumptions about the interconnect and other associated latencies on the die as it could take many cycles to send a signal from one end of the chip to the other. Clustering the cache was the first step in eliminating one such long latency. Another source of centralization is the instruction fetch and dispatch stages. We assume that these stages continue to remain centralized (since we are working with a single process and thread of control), but model the fact that the various clusters could be at different distances from the dispatch stage. Hence, the closest cluster could receive the renamed instruction in a single cycle after dispatch, while the furthest could take many cycles to receive its instruction. This effectively increases the number of front-end stages for some of the instructions and we later show the resulting performance impact.

The various clusters can be connected by a bus, crossbar, or a ring interconnect [1] for communicating values among clusters. A bus interconnect is limited by contention for its shared bandwidth and does not scale well as the number of clusters are increased. A crossbar interconnect involves more hardware especially as the number of clusters is increased. We therefore use two uni-directional ring interconnects in our simulations — while the cross-sectional bandwidth is lower than that for a crossbar, the latencies for neighboring clusters are also lower.

To evaluate scalability, we start with the large monolithic processor in Table 1 and break it down successively into 2, 4, 8, and 16 clusters. The smallest processor core in the 16 cluster case has 25 int and 25 fp registers, 10 integer issue queue and 10 fp issue queue entries, and a small 8KB cache bank. Although it can issue up to four instructions each cycle, this is restricted to one to each type of functional unit (integer ALU, integer mult-div, floating point ALU, floating point mult-div). We refer to this small core as X and then examine performance for

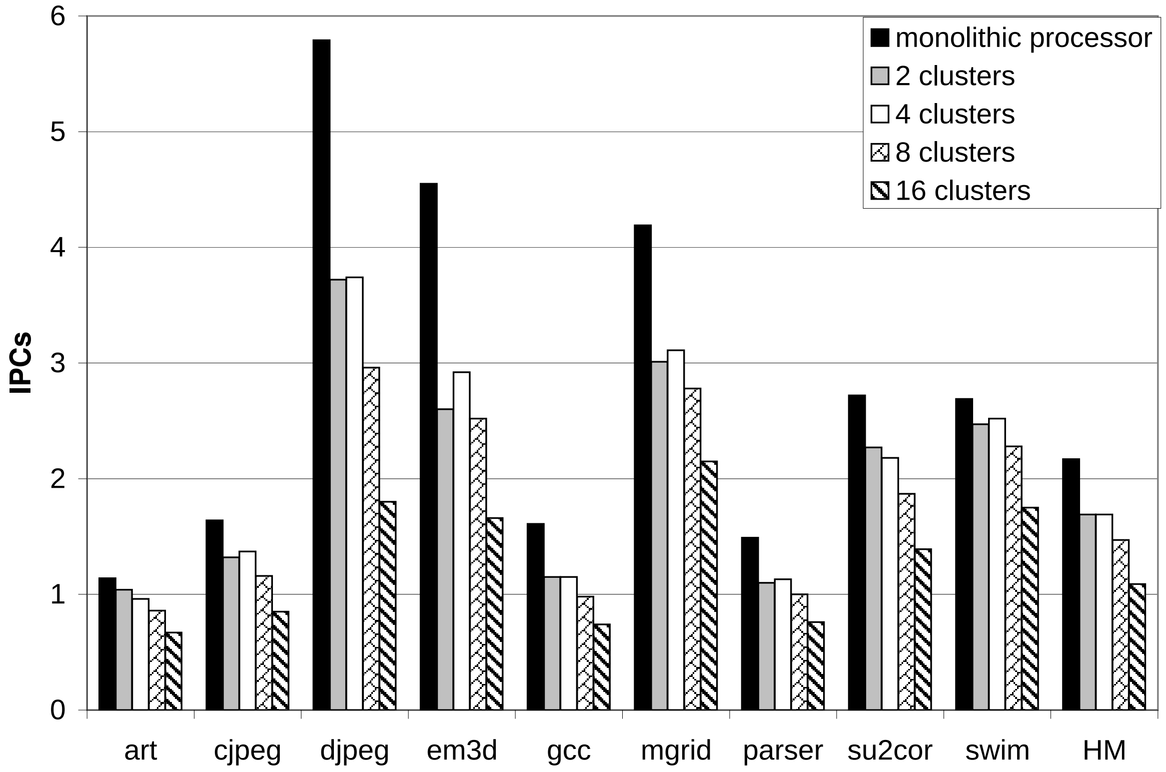

Figure 3. IPCs for processor models with the same resources, but increasing numbers of clusters.

processors $2X$, $4X$, $8X$, and $16X$, where $2X$ has two X clusters, and so on.

At the outset, we first examine the effect of assuming asymmetric latencies while sending instructions from the centralized dispatch stage to their respective clusters. We assume that the front-end lies in the center of the clusters, and that dispatch incurs a multi-hop delay proportional to the distance from the center. Thus, in the case of 16 clusters, the furthest cluster is eight hops away. Even for this extreme case, the IPC degradation from modeling this was only 5%. Hence, a decentralized implementation of the front-end would not yield a benefit of more than 5%. On the other hand, each additional cycle to the cache access time causes an IPC degradation of about 4%, making its decentralization necessary, especially for larger processors at future technologies.

Figures 3 and 4 demonstrate scalability along the two axes. In the first figure, starting with the monolithic case, the processor is broken into a number of clusters. Using two or four clusters causes an IPC degradation of about 22% (compared to the monolithic case). Going to eight and 16 clusters takes this number up to 32 and 50% respectively. The IPC degradation in going from 8 to 16 clusters is as much as 26%, requiring that the reduction in cycle time by halving the sizes of the resources would have to be more than 26% for improved performance. The second figure shows the effect of adding more resources in the form of more clusters. There is a big jump in IPC as the initial clusters are added. But in going from $8X$ to $16X$, there is only a 1% increase in IPC.

To better understand the causes for these trends, we study the contributions to IPC loss from various sources. There are four primary bottlenecks to performance when a monolithic processor is clustered. (i) Communication stalls from register dependences (*regstalls*), (ii) Communication stalls arising from bank mispredictions for loads (*cachestalls*), (iii) Issue stalls arising from the increased contention for limited resources in each cluster (*issuestalls*), (iv) Dispatch stalls because register replication leads to fewer effective registers and dispatch is stalled as soon as a cluster gets full (*dispstalls*). Figure 5(a) shows the HM of IPC degradation (relative to the monolithic

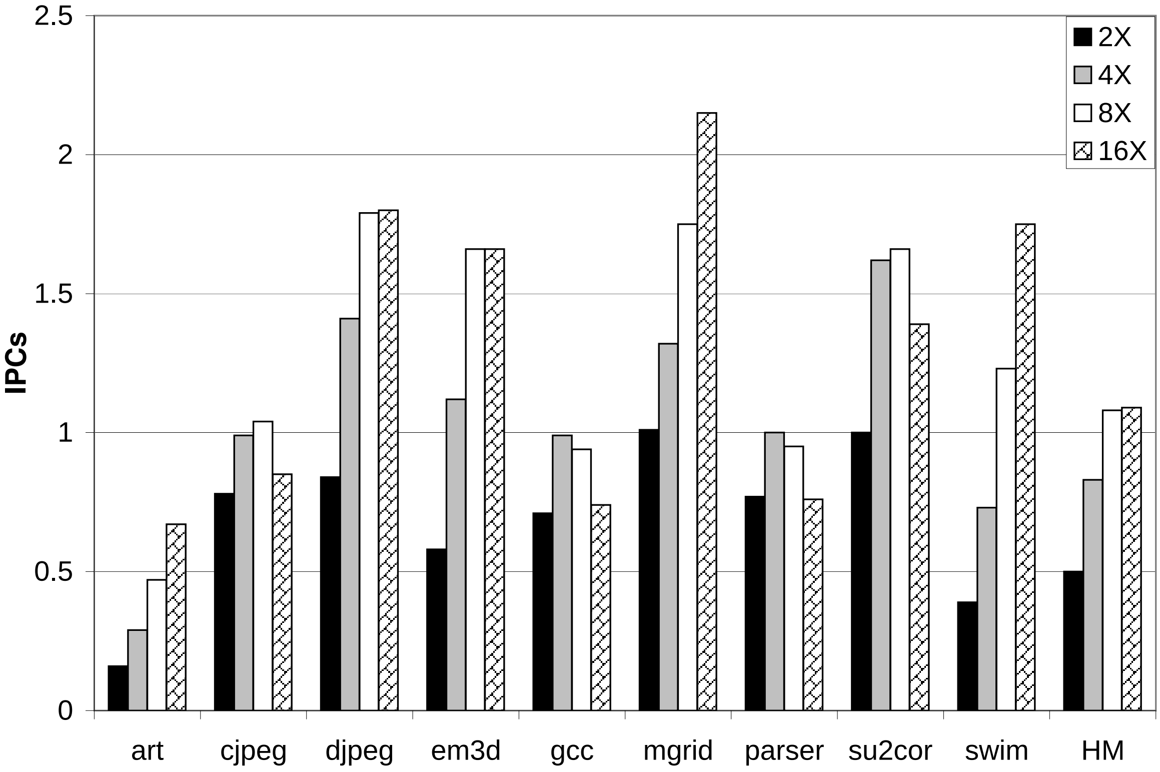

Figure 4. IPCs for varying numbers of clusters, with the cluster core being the same in each case.

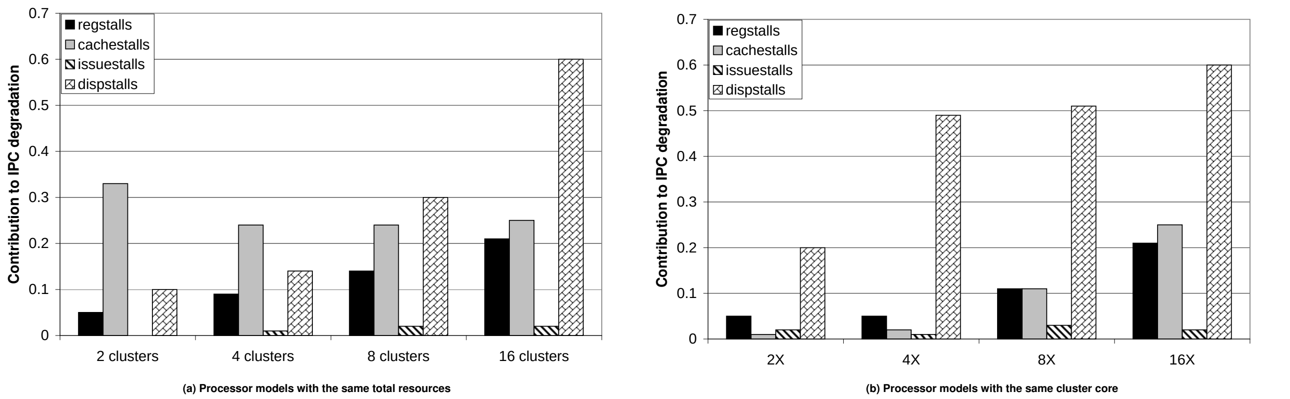

Figure 5. Contributions to IPC degradation from various sources. (a) With total processor resources being the same, different number of clusters. (b) With the cluster core being the same, different number of clusters.

processor with the same resources) caused by each factor (computed by turning each effect off and noticing the IPC change). In the first figure, the sum of the *regstalls* and *cachestalls* is similar for the 2-clustered and 4-clustered cases. While the 4-clustered case results in more total communication with potentially higher latencies (because of multiple hops), the communication bandwidth is cut in half for the 2-cluster case, resulting in more stalls due to increased interconnect contention.

In general, it can be seen that most of the performance degradation in all cases occurs because of stalls from register dependence and cache bank mispredictions, or from stalls to the dispatch stage. Focusing on Figure 5(b), where the per-cluster resources are kept constant, and performance evaluated as the number of clusters is increased, we can see that all three effects increase as the number of clusters is increased. The relative impact of *dispstalls* dominates as demonstrated in both Figure 5(a) and (b). For each of these cases where the number of dispatch stalls plays a large role, the size of each individual cluster is very small – it has as few as 25 integer and 25 fp registers, and 10 integer and 10 fp issue queue entries. In the case where there are as many as 16 clusters, the greater need for cross-cluster communication increases the degree of register replication, further exacerbating the dispatch problem. Clearly, the improved dispatch of instructions is of great importance as we scale to more aggressive clock speeds and hence smaller clusters.

The second observation we make is that for a number of programs, going from $8X$ to $16X$ (Figure 4) causes a drop in performance. As we increase the number of clusters, data values get spread out and the number of hops between clusters also goes up, thereby increasing the communication cost, as is evident from the increase in *regstalls* and *cachestalls* in Figure 5(b). The bank prediction accuracies for $2X$, $4X$, $8X$, and $16X$ are 84, 68, 58, and 45% respectively. For some of the programs, the increase in communication cost offsets the advantage of having more resources and a larger in-flight window.

6 Alleviating the Bottlenecks

The study in the earlier section revealed the two biggest bottlenecks in scaling the clustered design. As clock speeds become more aggressive and the size of each cluster decreases, dispatch stalls become very frequent. As the number of clusters increase, the degree of register replication increases, further exacerbating dispatch. In addition, register and data values are distributed over a wider space, thereby increasing the communication stalls. In this section, we first evaluate three mechanisms that help alleviate the dispatch problem. Finally, we propose a mechanism that reduces the communication stalls for a highly clustered processor.

6.1 Altering the Dispatch Policy

Given that the dispatch stage becomes a bottleneck in the future, we re-examine our dispatch policy. If an instruction being dispatched is assigned by the steering heuristic to a cluster that is full, we evaluate two different policies. The first chooses to stall the dispatch stage altogether. The expectation is that the cluster will soon free up resources and dispatch can proceed. The second steers the instruction to the nearest cluster that has available resources if the assigned cluster is full. Dispatch is stalled only if all clusters are full. Stalling dispatch avoids the communication costs incurred by the latter. Figure 6 compares the IPCs for the model where dispatch is stalled when the assigned cluster is full (dotted line) against a model where an instruction is steered to the nearest cluster if the assigned cluster is full (solid line). We see that for fewer clusters, the latter policy performs worse because, while it allows larger in-flight windows, it also increases the communication cost. As we move to more clusters, however, with dispatch becoming the bottleneck, the advantage from using a larger in-flight window is greater. When using 16 clusters, the policy that steers instructions to neighboring clusters performs 4% better than the one that stalls dispatch. The corresponding improvement in the $8X$ case is 5%.

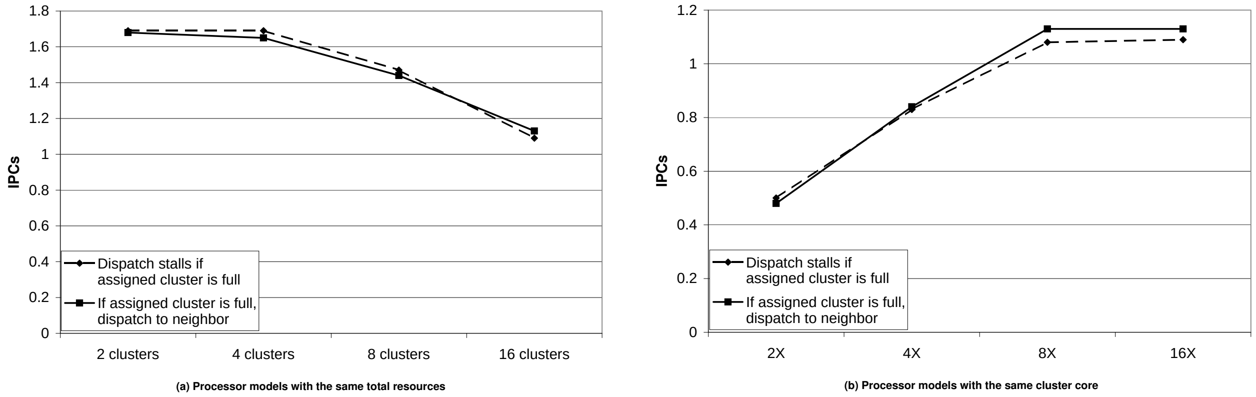

Figure 6. IPCs for various cluster organizations. The dotted line shows a dispatch policy where dispatch is stalled if the instruction is being assigned to a full cluster. The solid line dispatches the instruction to the neighboring clusters if the assigned cluster is full.

6.2 Reducing Register Replication

One of the major causes for the frequent stalling of the dispatch stage is the high degree of register replication. When an instruction sources a value produced in a physical register, PR_a, in another cluster, it creates another physical register, PR_b, in its own cluster. This is noted in the register map table and in the ROB entry for the consuming instruction. When the logical register corresponding to PR_a is remapped by another instruction, the ROB entry for this instruction includes PR_a and PR_b as physical registers that need to be freed when the instruction commits. On a branch mispredict, while squashing instructions along the incorrect path, the replicated physical registers created by the instructions are also freed.

While such a register allocation and deallocation policy is easy to implement, it is also extremely inefficient. Many register values that have already been consumed are kept around until the commit process for the sake of correctness, *i.e.*, they might still be used in the event of a branch mispredict or exception. However, the replication of such registers is a needless waste of valuable resources. Keeping only one of these values would be enough to ensure correctness.

To determine which registers can be deallocated early, we maintain a table to keep track of consumers. This table is located near the register map table and has a matrix of counters. Each counter has a physical register name associated with it. All counters along a row represent copies of the same physical register (i.e., copies of a value assigned to a logical register by a single store). Some of the rows also have the corresponding logical register associated with them if the physical registers in that row are the latest mappings for that logical register. As physical register copies are made for the logical register, they are added to that row. At the time of register rename, each consumer of a physical register increments its counter. When the instruction issues or gets squashed, it decrements the same. When the counter for a physical register becomes zero, it can be freed so long as there are other valid copies for that register. To make this decision logic simple, the various counters for the same value are physically placed in a single row. When a physical register is freed in this manner, its presence in the ROB is also removed. At every branch, the register map table is checkpointed. The presence of the physical register in these checkpoints is also removed. This can be easily implemented if the register map table is a CAM structure that maps physical registers to logical register [21] – the entry is simply cleared.

Most register values are only read once [12]. Given the high degree of register replication, the lone consumer of a register value is often sent to a different cluster. In this case, the number of consumers for the physical register mapped to the producer is always zero and it can be freed early. If the instruction has already issued, an extra bit in the table is set to indicate that the register has not been written, and cleared on completion. If the instruction is still in the issue queue, it is tagged to indicate that the result must be forwarded to a different cluster, and the

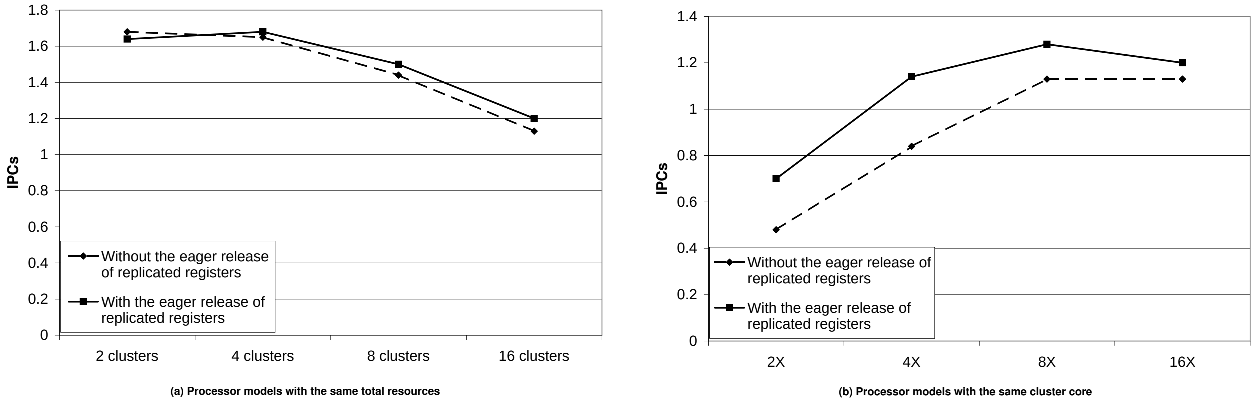

Figure 7. IPCs for various clustered organizations. The dotted line represents a model that does not eagerly free replicated registers, while the solid line represents a model that does.

physical register can be freed.

Figure 7 shows the effect of the eager register release mechanism on the various organizations. The base case includes the improved scalable dispatch policy from the earlier subsection. For the processor models that are limited by the dispatch stage ($2X$, $4X$, $8X$, $16X$), the IPC improvements are great - 46%, 36%, 13%, and 6%, respectively. In the 16-cluster case, the average integer register replication goes from 173 to 36, causing the average in-flight instruction windows to go from 209 to 280. The slight IPC degradation seen for the large processor with two clusters is a result of noise caused by a phenomenon explained in the next subsection.

In the clustered model described by Zyuban and Kogge [30], every time an instruction sources a register produced in a different cluster, it creates an entry in a buffer (which could be regarded as a separate bank in the register file) and once the consuming instruction reads the value in the buffer, it gets freed. While this implementation is a lot simpler, it has a couple of drawbacks - (i) two consuming instructions in the same cluster would have two replicated registers instead of one, thereby increasing register pressure, (ii) the physical register that is mapped to the producer is never freed early. The first factor is probably infrequent, but the latter would be an extremely common case. As already noted, when registers are replicated, the register in the producing cluster often has no consumer and can be freed right away. To make this possible, we had to ensure that there was a valid mapping of this value in another cluster, thus requiring the additional hardware at the rename stage. To disallow the producing instruction from writing into its earlier mapping, we had to introduce additional tag bits.

6.3 Intentional Dispatch Stalls

Once the registers or issue queue entries in the different clusters get consumed, dispatch gets stalled. Instructions can continue to dispatch when resources in clusters get deallocated. In a cycle, only the resources in some clusters may get deallocated and the newly dispatched instructions would be forced in to those clusters. Thus, the steering of an instruction is governed not by its own dependences, but by the order of resource deallocation caused by the commit/issue of instructions unrelated to it. To alleviate this problem, we intentionally introduced additional stalls in the dispatch stage whenever it got stalled. Every stall resulted in the dispatch stage being inactive for the next 20 cycles (experimentally determined to be the best performing number). In this period, resources from a number of clusters would have deallocated and once dispatch resumes, the steering heuristic can proceed as before. Figure 8 shows the IPC benefit from adding this policy to a base case that already incorporates the other proposed mechanisms in this section. In most cases, the benefit is an additional 3-4%. In the $2X$ case, this policy performs slightly poorly as the communication cost from steering to the wrong cluster is small and dispatch is already so constrained that stalling it further degrades performance.

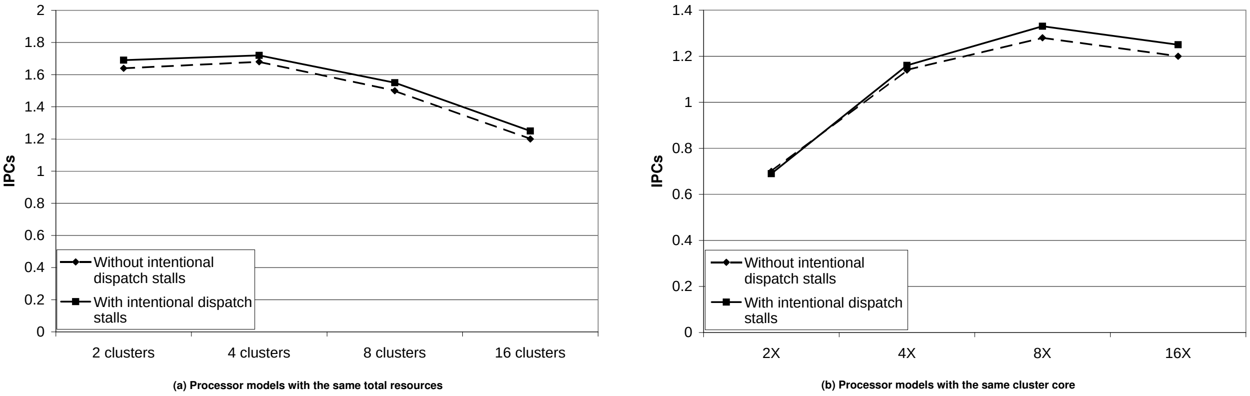

Figure 8. IPCs for various clustered models. The dotted line represents a processor model that does not create artificial stalls in dispatch. The solid line represents a model that does.

6.4 Using Fewer Clusters

In Figures 7(b) and 8(b), the processor with $8X$ resources outperformed the processor with $16X$ resources. Adding more resources to the processor need not necessarily improve performance. Supporting a larger window improves performance, but at the same time, the various register and cache values are more distributed in the processor, thereby increasing the communication costs. Different programs behave differently depending on how much they benefit from exploiting far-flung ILP. In the earlier unoptimized clustered cases in Figure 4, when dispatch was a frequent bottleneck, adding more resources often led to improved performance. But once the dispatch bottleneck has been alleviated, adding more resources results in an overall performance loss. Given that future processors might have the ability to have as many as 16 clusters on a die, we examine if the number of active clusters can be dynamically tuned to suit the application's requirements.

If the processor has 16 clusters connected by a ring interconnect, using only 8 clusters does not reduce the worst-case latency. The two most distant clusters in the processor would still be eight hops away – this is similar to a layout where all clusters are laid out in a straight line. Even so, such an organization does reduce the average communication latency and would be useful for programs that did not benefit from large in-flight windows. While disabling some of the clusters, the dirty data in the caches of all clusters would have to be flushed to the L2. The line size for the caches would also be reduced, thus altering the indexing function. As a result, such changes would be feasible only at a reasonably coarse granularity of about a million or so instructions. Algorithms like that described in [3], which try out various configurations at the start of each program phase and pick the best performing one, would apply here as well. While we do not show results for such a dynamic choice of cluster configurations, in Figure 9, we show the effect of various static organizations. The first four bars show IPCs for the cases $2X$-$16X$ (with the earlier optimizations in this section incorporated), the fifth bar shows a 16-cluster organization with only eight clusters active and the sixth bar shows the same 16-cluster organization, but with only four clusters active. For a number of programs, the $8X$ organization outperforms the $16X$ organization (cjpeg, djpeg, em3d, gcc, parser, su2cor). In fact, for three of the most branch predictor limited programs (cjpeg, gcc, parser), the $4X$ organization performs best. Because of the high branch mispredict rate, having a large in-flight window is often not beneficial for such programs. We see that for these programs, the 16-cluster organization with fewer active clusters outperforms the 16-cluster case. With eight active clusters, speedups of 9%, 15%, 13%, and 4% over the 16-cluster case are seen for cjpeg, gcc, parser, and su2cor, respectively. Note that this organization cannot quite achieve the same performance as the $8X$ case as the broken ring structure causes it to have a higher communication latency on average. For example, in gcc, the average communication latency per instruction is 1.8 for the $8X$ case, 2.9 for the $16X$ case, and 2.6 for the $16X$ organization with eight active clusters. With four active clusters, the speedups for these three programs over the 16-cluster case are 23%, 33%, 28% and 6%. Assuming

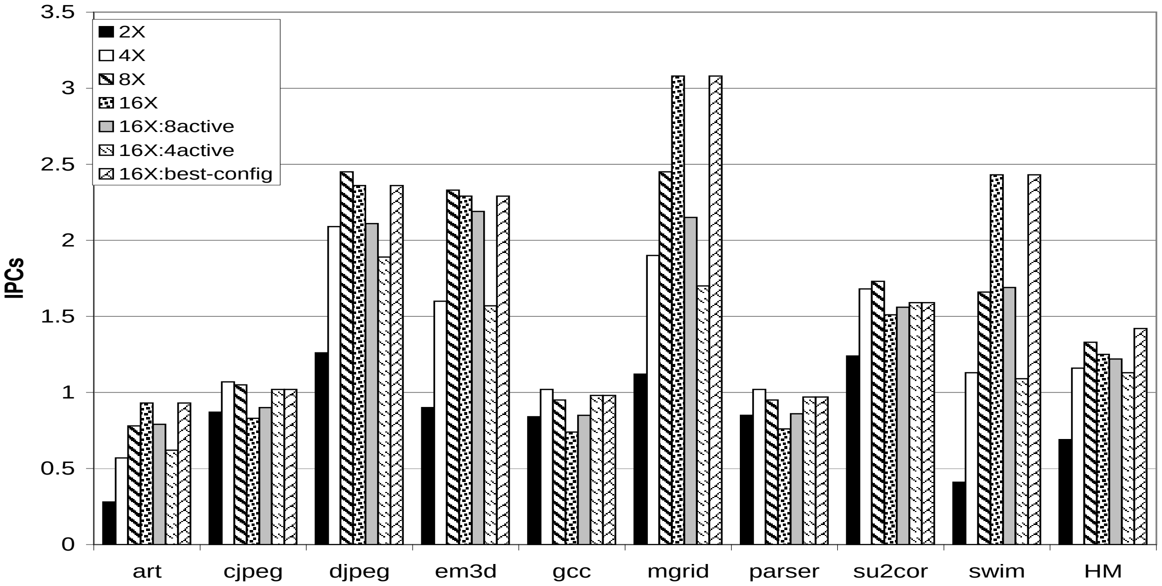

Figure 9. IPCs with the identical cluster core, but with varying number of clusters from 2 to 16. The next two bars represent a model with 16 clusters, of which only some clusters are active. The last bar shows a model with 16 clusters, but which uses only either 4, 8, or 16, depending on the program.

that the processor has 16 clusters, and a dynamic selection algorithm is able to pick the best configuration for each program (among $16X$, $16X : 8active$, and $16X : 4active$), the overall IPC would be 1.42 (the last bar in Fig 9), a 14% improvement over $16X$, and a 7% improvement over $8X$.

6.5 Summary

For the heavily clustered processor model ($16X$), we showed that various mechanisms that alleviate the dispatch and communication bottlenecks can greatly improve its performance. When a cluster is full, dispatching to neighboring clusters instead of stalling dispatch improves performance by 5%. Eagerly freeing up register values results in an additional 6% improvement, and purposely introducing dispatch stalls when all clusters are full adds a further 4% improvement. Finally, by selectively using fewer than the maximum possible clusters for certain programs, we achieved an additional 14% improvement. Thus, the overall improvement over the base 16-clustered processor (which includes a clustered cache) from all these changes was 30%. The improvements on individual programs are shown in Fig 10. This makes the 16-clustered processor model an attractive design point – in going from an 8-clustered processor to a 16-clustered processor with the same resources, the drop in IPC is only 8%, when it had been 26% without the proposed mechanisms in place. This degradation can now more easily be overcome by the much faster clock expected with the 16-cluster design.

7 Related Work

The idea of clustering has been around for a while. The Multiflow architecture [18] and the Limited Connectivity VLIW [9] were among the earlier works that distributed the register file among groups of ALUs. Farkas et al [13] deal with a dynamically scheduled processor and also distribute the issue queue across the clusters. They

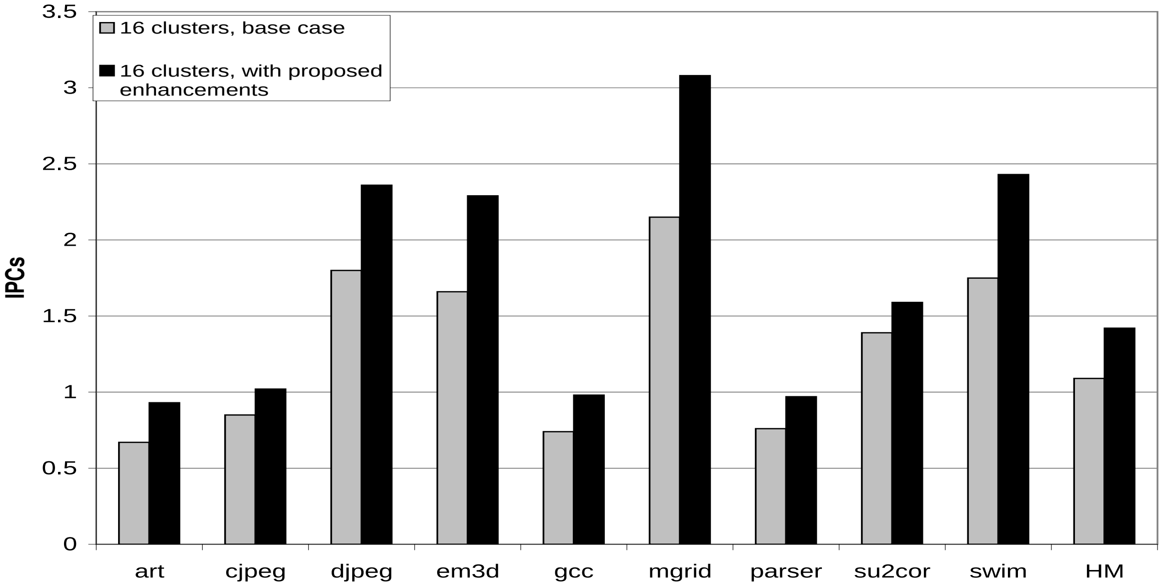

Figure 10. IPCs for the base 16 clustered processor and for a processor model that includes the proposed enhancements to the dispatch policy, the eager release of registers, and the disabling of clusters on a per-application basis.

do a compile-time assignment of instructions to clusters by looking at neighboring static instructions. Canal et al [7, 8] use run-time statistics to do a dynamic steering of instructions to clusters. The best performing heuristic uses information on register dependences and load imbalance. Baniasadi and Moshovos [5] do a similar study for a quad-clustered processor model. Their results show that a simple heuristic that assigns a series of n instructions to a cluster, where the cluster is picked in a round-robin manner, performs as well as other more complicated heuristics that take dependences and past history into account. The Alpha 21264 [16] is a clustered microarchitecture with two clusters, but replicates the entire register file. Ranganathan and Franklin [22] cluster the functional units and the issue queue, but maintain a centralized register file.

Various other architectures like Multiscalar [26], Trace processors [25], and DMT [2] have been proposed that use multiple execution units (clusters). Each cluster is assigned a sequential segment of the program, with all except one of them being speculative in nature. Palacharla [21] proposes a clustered issue queue, where successively dependent instructions are steered into a single cluster. The head entries of each queue are the only instructions that could be made ready in a cycle, hence the complexity of the wakeup and select logic is greatly reduced.

All these various bodies of work do not attempt to distribute the cache among the various clusters. Recent studies have looked at the problem of increasing cache bandwidth, but have not placed them in the context of clustered microarchitectures. A truly multi-ported cache would be very expensive to implement, prompting the proposal of various alternatives. One such alternative is double-pumping, which runs the cache at a frequency that is an integral multiple of the processor frequency. Since this technique has limited scalability, banked caches have emerged as the acceptable means of providing high bandwidth [23]. Rivers et al [23] discuss the trade-offs of various designs. Replicating the data cache, with each bank having a single read and write port, can accommodate multiple reads in the same cycle, but only a single write can occur in a cycle as it would have to write simultaneously to each bank. In a line interleaved cache, each bank gets a subset of the total cache lines. Multiple accesses can be supported in a cycle so long as they are to different cache lines. This restriction degrades IPC somewhat as spatial locality dictates that successive accesses are often to the same cache line. Word interleaving tries to alleviate this problem by distributing each cache line across the various banks. This implies that, for example, in a dual-banked cache, at most one odd and one even word can be accessed in a cycle. However, each bank would have to replicate the tag storage, unlike in a line interleaved cache. The banked organizations cluster the cache, but leave the rest of the processor, including the load-store queue (LSQ) untouched.

Cho et al [11, 10] cluster the cache and the LSQ, but not the rest of the processor. This helps reduce the size of the LSQ and also serves to increase the bandwidth of the cache system without true multi-porting. They use a different criterion for splitting the cache into banks. They notice that 50% of all data accesses are made to the stack/frame and this can often be detected by just looking at the instruction. Only in rare cases can a load be ambiguous in terms of which data region it is accessing. Loads and stores are split into one of two streams early in the pipeline. They make the optimistic assumption that stores in one stream will not conflict with loads in the other stream. As a result, loads in one stream need not compare their address with the addresses of pending stores in the other stream, and this helps reduce the size of the LSQ. One of the cache banks only contains stack and frame data, while the other bank maintains the rest of the data, allowing one access to each bank per cycle. The authors further make the observation that due to the small size of temporary data allocated on the stack, the cache bank servicing the stack accesses can be as small as 4KB. At the time of actually issuing the ld/st, if it is discovered that the instruction was miscategorized, recovery is initiated. The authors do not address the problem of how such a partition would scale beyond two clusters.

Yoaz et al [29] anticipate the importance of splitting data accesses across multiple streams early and propose predictors to do the same. They evaluate the accuracy of branch predictor-like schemes to do the prediction and find them to be about 70% accurate for two streams. They propose two uses for such a predictor - (i) improved scheduling of memory operations, and (ii) the use of multiple memory pipelines like that proposed by Cho et al [11, 10]. They mention that predictions with low confidence can direct the instruction to all the memory pipelines.

The works described so far have either tried to cluster the issue queue, register file, and the functional units, or the LSQ and data cache. Recently, an organization that combines the two was proposed by Zyuban and Kogge [30] and is very similar to our implementation of the clustered cache. However, the proposed organization was only evaluated from the point of view of power consumption, with the conclusion being that a clustered organization is much more power efficient than a centralized one. Our paper is the first that quantifies the IPC effect of clustering the cache.

Another recent study by Aggarwal and Franklin [1] explores the performance of various steering heuristics as the number of clusters scale up. Theirs is the only study that looks at as many as 12 clusters and proposes the use of a ring interconnect. They conclude that the best steering heuristic varies depending on the number of clusters and the processor model. To take this into account, each of our clustered organizations was optimized by tuning the various thresholds in our steering heuristic. Their study assumes a centralized cache.

Our condition for identifying replicated registers that can be freed is similar to that used by Balasubramonian et al [4] while identifying registers to be copied into a second level register file.

8 Conclusion

The current trends towards faster clocks, shrinking process technologies, and increasing transistor budgets necessitates the design of a clustered processor that has many small cores. In this paper, we examine the scalability of such a design, identify the bottlenecks, and propose various novel enhancements. Some of the salient contributions made are -

- The evaluation of a clustered cache and bank predictor incorporated with the clustering of the register file, issue queue, and functional units.

- A detailed study of scalability aspects of a clustered microarchitecture, while taking a clustered cache and multi-cycle dispatch latencies into account.
- Novel modifications to the dispatch policy that help alleviate the dispatch bottleneck for highly clustered processors.
- The use of an eager register release policy to reduce the degree of register replication.
- The proposal and evaluation of a clustered processor that changes the number of active clusters depending on the application's requirements.

Before adding our various proposed mechanisms, the clustered microarchitecture displayed poor scalability. With the enhancements in place, the overall performance of the 16-cluster organization improved by 30%. In going from an 8-cluster processor to a 16-cluster processor with the same total resources, the IPC drop is now a much more tolerable 8% (while it had been 26% in the base clustered model), thereby permitting the 16-cluster alternative to achieve better performance via a faster clock.

References

[1] A. Aggarwal and M. Franklin. An Empirical Study of the Scalability Aspects of Instruction Distribution Algorithms for Clustered Processors. In *Proceedings of ISPASS*, 2001.

[2] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. In *Proceedings of MICRO-31*, pages 226–236, 1998.

[3] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. In *Proceedings of MICRO-33*, pages 245–257, Dec 2000.

[4] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Reducing the Complexity of the Register File in Dynamic Superscalar Processors. In *Proceedings of MICRO-34*, Dec 2001.

[5] A. Baniasadi and A. Moshovos. Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduled, Superscalar Processors. In *Proceedings of MICRO-33*, pages 337–347, Dec 2000.

[6] D. Burger and T. Austin. The Simplescalar Toolset, Version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, June 1997.

[7] R. Canal, J. M. Parcerisa, and A. Gonzalez. Dynamic Cluster Assignment Mechanisms. In *Proceedings of HPCA-6*, 2000.

[8] R. Canal, J. M. Parcerisa, and A. Gonzalez. Dynamic Code Partitioning for Clustered Architectures. *International Journal of Parallel Programming*, 29(1):59–79, 2001.

[9] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned Register Files for VLIWs: A Preliminary Analysis of Trade-offs. In *Proceedings of MICRO-25*, 1992.

[10] S. Cho, P.-C. Yew, and G. Lee. Access Region Locality for High-Bandwidth Processor Memory System Design. In *Proceedings of MICRO-32*, pages 136–146, 1999.

[11] S. Cho, P.-C. Yew, and G. Lee. Decoupling Local Variable Accesses in a Wide-Issue Superscalar Processor. In *Proceedings of ISCA-26*, pages 100–110, 1999.

[12] J.-L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham. Multiple-Banked Register File Architectures. In *Proceedings of the ISCA-27*, pages 316–325, 2000.

[13] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing Cycle Time through Partitioning. In *Proceedings of MICRO-30*, 1997.

[14] B. Fields, S. Rubin, and R. Bodik. Focusing Processor Policies via Critical-Path Prediction. In *Proceedings of ISCA-28*, July 2001.

[15] L. Gwennap. PA-8500's 1.5M cache aids performance. *Microprocessor Report*, 11(15), November 17, 1997.

[16] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[17] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of MICRO-30*, pages 330–335, 1997.

[18] P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. O'Donnell, and J. Ruttenberg. The Multiflow Trace Scheduling Compiler. *Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[19] D. Matzke. Will Physical Scalability Sabotage Performance Gains? *IEEE Computer*, 30(9):37–39, Sept 1997.

[20] G. McFarland. *CMOS Technology Scaling and Its Impact on Cache Delay*. PhD thesis, Stanford University, June 1997.

[21] S. Palacharla, N. Jouppi, and J. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of ISCA-24*, 1997.

[22] N. Ranganathan and M. Franklin. An Empirical Study of Decentralized ILP Execution Models. In *Proceedings of ASPLOS-VIII*, pages 272–281, 1998.

[23] J. Rivers, G. Tyson, E. Davidson, and T. Austin. On High-Bandwidth Data Cache Design for Multi-Issue Processors. In *Proceedings of MICRO-30*, pages 46–56, 1997.

[24] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting Dynamic Data Structures on Distributed Memory Machines. *ACM TOPLAS*, Mar 1995.

[25] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace Processors. In *Proceedings of MICRO-30*, 1997.

[26] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *Proceedings of ISCA-22*, 1995.

[27] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of ISCA-22*, pages 392–403, 1995.

[28] E. Tune, D. Liang, D. Tullsen, and B. Calder. Dynamic Prediction of Critical Path Instructions. In *Proceedings of HPCA-7*, Jan 2001.

[29] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation Techniques for Improving Load Related Instruction Scheduling. In *Proceedings of ISCA-26*, pages 42–53, 1999.

[30] V. Zyuban and P. Kogge. Inherently Lower-Power High-Performance Superscalar Architectures. *IEEE Transactions on Computers*, Mar 2001.