In Proceedings of the 18th International Parallel and Distributed Processing Symposium, April, 2004

Clustered Multithreaded Architectures – Pursuing Both IPC and Cycle Time

Jamison D. Collins Dean M. Tullsen
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114

Abstract

Clustering is an architectural technique that allows the design of wide superscalar processors without sacrificing cycle time, but at the cost of longer communication latencies. Simultaneous multithreading architectures effectively tolerate instruction latency, but put even more pressure on timing-critical processor resources. This paper shows that the synergistic combination of the two techniques minimizes the IPC impact of the clustered architecture, and even permits more aggressive clustering of the processor than is possible with a single-threaded processor.

Additionally, this paper shows that multithreading enables effective instruction steering policies unavailable to a single-threaded clustered architecture. This paper explores the impact of aggressively clustering four complex processor structures, (1) instruction window wakeup and functional unit bypass logic, (2) register renaming logic, (3) the fetch unit, and (4) the integer register file, on a simultaneous multithreading processor.

1. Introduction

Processor architectures have traditionally achieved performance gains through a combination of clock speed acceleration and improvements in instruction level parallelism. The resulting performance of a new processor generation scales with the product of these two factors. Increasingly, however, those factors can no longer be considered independent. Wide superscalar execution implies both wide and complex control and storage structures, all of which put significant upward pressure on the processor cycle time [16]. Hardware clustering is a processor architecture technique that allows a wide superscalar processor to still maintain an aggressive clock rate. It groups pipeline resources into smaller clusters, allowing local communication to travel shorter distances, and allowing complex control logic to operate on a subset of the total processor resources. However, these architectures still incur costs, particularly the increased latency of non-local communication, which reduce the achieved instruction throughput. For example, the Compaq Alpha 21264 [11] clusters the functional units into two sets. Operand bypassing is fast within a cluster, but slower between clusters. A hardware clustered architecture is a compromise between a chip multiprocessor and a monolithic superscalar. Like the chip multiprocessor, timing-critical resources are distributed, but unlike the multiprocessor, some resources (not timing critical) remain shared for higher efficiency.

We expect this trend toward hardware clustering to continue, and advance in three directions – deeper clustering (more pipeline functions clustered), heavier clustering (more clusters), and higher inter-cluster communication costs. This work will focus on the first trend, recognizing that as we move along the other two directions, the results and principles shown in this paper will be amplified.

Simultaneous multithreading (SMT) architectures [25, 24, 27, 10, 14] have the ability to execute instructions from multiple threads each cycle. These architectures effectively tolerate virtually all latencies observed by instructions in the pipeline. The additional latencies incurred by a clustered architecture are no different.

Multithreading is highly synergistic with clustering for three reasons. First, multithreading motivates the design of wide superscalar processors. While it is not clear that an 8-issue general-purpose single-threaded processor would be viable due to the lack of inherent ILP in most applications, an SMT processor can easily take full advantage of an 8-issue, 16-issue, or wider processor – if we can build them without sacrificing clock speed. Second, multithreading can in some cases put even more pressure on timing-critical structures, such as the register file and possibly the rename logic. Third, this work will show that multithreading also hides the latencies introduced by clustering, enabling more aggressive clustering configurations with smaller performance (instruction throughput) costs.

A clustered multithreaded architecture, then, has the ability to exploit both thread-level parallelism and instruction-level parallelism, yet still maintain high clock rates to maximize overall throughput. Neither clustering or multithreading preclude traditional multiprocessing on the chip, which might include multiple clustered, multithreaded processors.

For a clustered processor to outperform a non-clustered architecture, performance (measured, for example, in instructions per cycle) must not be degraded linearly with in-

creased communication costs between clusters. We show that multithreading can make performance much less sensitive to those costs than a conventional architecture. An SMT processor also provides another dimension of freedom (i.e., by incorporating the thread id) in devising instruction scheduling algorithms to mitigate the effects of clustering delays, and those algorithms are a primary focus of this research.

This paper examines hardware configurations that cluster the instruction queues, rename logic, fetch unit, and the register file. We demonstrate techniques to minimize the negative performance effects of clustering in each of these environments, in the context of a hardware multithreaded architecture. Despite the wide variety of clustering architectures observed, several key principles are demonstrated:

First, SMT processors hide much of the latency incurred by clustering. This ensures that a larger portion of the clock rate increases enabled by clustering are translated into real performance gain. This can also enable more aggressive clustering approaches than might have been feasible without multithreading. Thus, multithreading enables the processor to be designed *more aggressively* for low complexity and high clock rate than a non-multithreaded architecture.

Second, an SMT processor in several cases enables the use of less complex instruction steering algorithms to achieve the same performance. This ensures that we don't replace the complexity of wide communication with other forms of complexity introduced by clustering.

Third, a clustered SMT processor design allows new instruction steering options that are either not feasible or not possible in a conventional architecture. We show that the best steering mechanisms are neither the same ones proposed for non-threaded architectures, nor the most obvious multithreaded mechanisms (static assignment of threads to clusters).

Fourth, the critical design issues of a multithreaded clustered architecture are very different from a conventional clustered architecture. The dominant performance phenomenon for clustered SMT processors is that of stalled shared resources, a phenomenon not exhibited in non-threaded architectures, and which significantly impacts the choice of steering algorithm.

Other techniques allow even single-thread workloads to tolerate clustering delays by utilizing available thread hardware. We demonstrate the effect of parallel compilation and a previously proposed technique for dynamically transforming a single threaded program into multithreaded.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 describes our simulation methodology. Section 4 describes the different clustered architectures we investigate. Section 5 presents the performance of these clustered architectures under a multiprogrammed workload. Section 6 describes the performance impact on single applications which have been multithreaded, and Section 7 concludes.

2. Related Work

Much previous research has evaluated the impact of clustering various processor resources. Baniasadi and Moshovos evaluate various queue clustering schemes for a single threaded processor [4]. This work focuses on queue assignment schemes when the instruction queue is split into four queues. Balasubramonian, Dwarkadas and Albonesi investigate the performance impact of clustering a single-threaded processor of up to 16 clusters in [3], and propose a dynamic scheme for selectively disabling clusters in order to reduce communication costs. Aggarwal and Franklin explore the use of different algorithms for assigning instructions to clusters in [1].

Palacharla, Jouppi and Smith study complexity and delay characteristics of processor structures as the processor issue width is varied [16]. They identify structures most likely to present the critical timing path for wide issue processors, and present results from dividing the traditional out-of-order queue into a series of instruction FIFOs to reduce complexity. Canal, Parcerisa and Gonzalez propose a number of queue clustering schemes [5]. The Alpha 21264 [9] clusters functional units and replicates register files.

Additionally, a number of highly partitioned processors have been proposed. Farkas, Chow, Jouppi and Vranesic propose the Multicluster Architecture [8], which divides functional units, registers and the instruction window between two clusters. They present a static instruction scheduling algorithm to handle dependences on values present in the other cluster. Rotenberg, Jacobson, Sazeides and Smith propose Trace Processors [17], which dynamically divide program instructions among multiple Processing Elements, each with private functional unites. Sohi, Breach and Vijaykumar propose Multiscalar Processors [20], in which software guides the division of programs into individual tasks, which are executed on small functional unit clusters known as processing units.

Krishnan and Torrellas study the tradeoffs of building multithreaded processors as either a group of single-threaded CMP cores, a monolithic SMT core, or a hybrid design of multiple SMT cores in [12]. However, they do not study the effects of allowing threads or instructions to migrate between different processor clusters, which is a primary focus of this paper.

3. Simulation Methodology

All results in this paper are generated with SMTSIM, a cycle accurate, execution driven simulator [22] that simulates an out-of-order, simultaneous multithreading processor. SMTSIM executes unmodified alpha binaries. The simulated processor has 64KB 4-way set associative L1 data and instruction caches, and an 8MB, 8-way set associative unified L2 cache. Latency to the on-chip L2 cache is 15 cycles and to memory is a further 250 cycles. We model a 2Bcgskew branch predictor, similar to the EV8 branch predictor [18], but without delayed ghr updates or shared hysteresis bits [18]. The predictor provides multiple predictions of

not taken branches or a single taken branch for each thread which fetches on a cycle.

We model an 8-wide processor with 8 integer functional units (two of which can perform loads and stores) and 3 floating point functional units. The processor contains an integer instruction queue of 128 entries, a 128 entry floating point instruction queue, 256 floating point and 256 integer registers available for renaming (in addition to the 128 registers to hold the logical register state of 4 hardware contexts).

When clustering the integer instruction queue, we model four smaller queues of 32 entries each (retaining an aggregate size of 128 entries). Because two load/store functional units cannot be evenly divided among four functional unit clusters, we allow one functional unit in each cluster to issue memory operations, but each cluster can only issue such instructions every other cycle. Thus, total L1 cache bandwidth remains the same as in the non-clustered processor.

Eight integer benchmarks from the SPEC 2000 suite are studied (`crafty`, `gcc`, `gzip`, `mcf`, `parser`, `twolf`, `vortex`, and `vpr`), all compiled with full optimization. Benchmarks run reference inputs but we skip the first five billion instructions before beginning detailed simulation. Benchmarks are then simulated for 300 million instructions times the number of threads. Threads are fetched using the ICOUNT fetch policy [24] whereby those threads with the fewest instructions at and before the queue stage in the pipeline are chosen to fetch. In all configurations up to two threads are fetched. In configurations with a single register renaming unit, the first thread fetches up to eight instructions, and then any remaining fetch bandwidth is used by a second thread. In configurations with two renaming clusters, up to four instructions are fetched from each thread.

In multithreaded configurations, multiple benchmarks are executed together. Each result in those sections represents the average of eight runs with different permutations of threads/applications, chosen such that each benchmark is equally represented in the average result. We report performance using the weighted speedup metric [23, 19]. Weighted speedup is computed as the sum of the speedups seen by individual threads (in a multithreaded execution) over the throughput they achieved in a baseline configuration. This metric prevents the experiments from achieving artificial speedups (when reported by IPC) simply by biasing execution toward threads that have particular characteristics, without necessarily improving global system performance.

All results reflect only the changes in instruction throughput, ignoring cycle time changes. Thus, results are expressed as slowdowns relative to a non-clustered architecture, even though most of those architectures would achieve speedups once the faster cycle time of the clustered architecture is accounted for. We do not model the exact impact of clustering on cycle time for several reasons. First, we are only striving to find the best architectural configuration within each clustering alternative – the cycle time is a constant across those alternatives. Second, cycle time effects are highly dependent on the specifics of the implementation, layout, and process. Even if our estimates were correct for a particular implementation, they would be wrong for others.

This research focuses on clustering of the integer execution pipeline for a workload of integer applications. This is merely to provide focus for the research. However, all of the principles identified in this study will apply to a clustered floating point pipeline running floating-point intensive code.

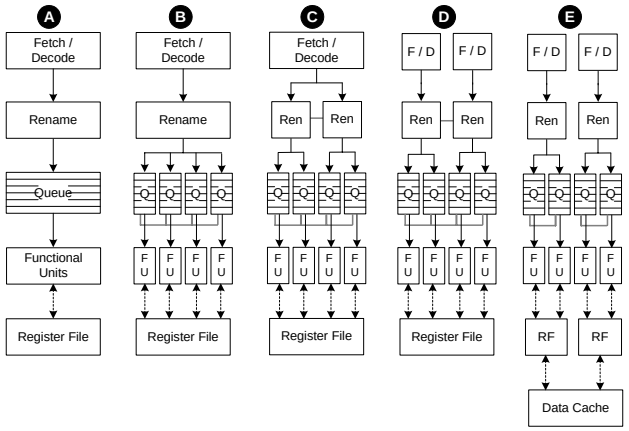

Figure 1. The five processor configurations explored in this research. Black lines represent instruction paths, gray lines added bypass paths added due to structure splitting, and dotted lines indicate register value movement. Non-split structures suffer no bypass penalty.

4. Impact of Splitting Processor Structures

This section describes the five processor configurations we explore, and the performance challenges presented by each configuration. The point of this research is not to compare these different architectures, but to recognize that each of these configurations may make sense within a technological window in the future; thus, we would like to understand how each of these architectures interacts with a multithreaded workload, and how to optimize performance. The architectures are shown in Figure 1, and moving left to right represents increasingly aggressive clustering. Notice that this work stops just short of considering chip multiprocessors [15], which are just one extreme end of the multithreaded/clustered design space. We ignore them here, not because they don't represent a reasonable part of the design space, but only because those architectures have been studied extensively, and their completely static partitioning provides fewer opportunities for the design optimizations which are the focus of this paper.

Each of these configurations represents a family of possible architectures with varied number of clusters and cross-cluster communication latencies. In each case, we have chosen to study a single representative that is reasonable in the near term, yet exhibits the important performance issues of that class of architectures.

The clustering configurations considered in this paper are as follows:

(A) The non-clustered processor Our baseline processor is a conventional SMT processor. All pipeline stages have

a bandwidth of eight instructions, and full value bypass is supported without additional delay.

(B) Clustered Execution Core The first structures we partition are the processor functional units and instruction queues. The central instruction queue is equally split into four smaller, 32-entry queues. Each instruction queue has exclusive access to two integer functional units (one capable of performing memory operations every other cycle) and which can bypass values between each other with no additional delays. The execution of a dependent instruction is delayed by two cycles when executing in a different cluster from a parent. This configuration incurs performance losses relative to the non-clustered processor from three sources:

(1) Dependent instruction execution delays — Because it takes two cycles to wake up a dependent instruction in a different queue cluster, the observed execution latency of the producing instructions can be effectively lengthened.

(2) Load imbalance — An instruction will not issue when all issue slots in its cluster have been allocated, even if resources are free in another cluster.

(3) Increased queue conflicts — attempting to insert an instruction into a full queue results in a queue conflict, preventing further instructions from entering queues and stalling preceding pipeline stages. This can occur even though entries may be available in other instruction queues.

(C) Register renaming clustering The second processor structure we cluster is the register renaming logic, which is reduced from a single, eight-wide register renaming unit into two four-wide units by splitting the dependent instruction comparator network and reducing the number of read ports on the map table. Both units still allocate renaming registers from a global free list as this simple structure is unlikely to represent the critical path.

When instructions are renamed in one renaming cluster, its map table is updated that cycle. The map table in the other cluster also receives this information, but with a delay. Attempting to rename instructions in a renaming cluster lacking that thread's most up-to-date renaming information forces a stall until the renaming information arrives. Keeping track of which thread can rename in which cluster is trivial, because the cluster selection decision is made from the non-clustered front-end. Processor performance is impacted in two additional ways from clustering this stage:

(1) Renaming bandwidth — The reduced-width renaming clusters can rename only four instructions per thread per cycle. Accordingly, processors with a clustered renaming unit utilize a slightly modified fetch policy in which the two threads with lowest ICOUNT value fetch up to four instructions each. When multiple threads execute, this restriction has a limited effect. However, single thread performance can be severely affected.

(2) Renaming conflicts — Renaming conflicts occur when some instructions cannot be renamed on a particular cycle. This occurs either because two instruction groups which reach register renaming simultaneously must both be renamed in the same renaming cluster, or because a group of instructions is attempting to rename in a cluster which does not yet have the thread's most recent renaming information.

(D) Front end clustering Front-end clustering divides the pipeline stages which precede register renaming (fetch and decode) into separate, four-wide pipelines. Each pipeline has its own private instruction queue entries and functional units (the register file, caches, and back-end pipeline stages remain shared). Splitting the pipeline in this way can actually be advantageous. Previously, a stall at any point in the pipeline (for example, due to an instruction queue becoming full) would cause instruction back-pressure and stall fetch entirely. With a split processor front-end, a stall causes back-pressure only in the pipeline in which it occurs. This can prevent a single poorly behaved thread from impacting the performance of the other threads in the processor, a problem noted in prior research [23]. However, the target pipeline must be decided at fetch time, reducing our flexibility in assigning instructions to different processor clusters. In addition to several of the previously discussed performance issues, a new form of renaming conflict is introduced with this form of clustering.

(1) Out-of-order renaming conflict — This is similar to the previous renaming conflict, except that (a) the later instructions can reach the renaming stage *before* the prior instructions are renamed, and (b) the later instructions could be stalled arbitrarily long waiting for the earlier instructions to be renamed (if the other pipeline is stalled).

(E) Register file clustering Having clustered the entire processor front-end and execution core, the register file remains the major shared structure. Even if it is replicated among execution clusters to reduce the number of read ports (as occurs in the Alpha 21264 [11]), the number of write ports and total number of registers required on each of these replicated register files remains large, likely putting register file access on the critical timing path. Thus, we seek a more aggressive mechanism for clustering the register file, for the purposes of this research. We split the register file by evenly partitioning registers between the two pipelines, with no replication of register values between the two.

Because there is no direct communication between register files, for a thread to switch pipelines its most recently produced register values must be copied to the pipeline it is switching to. We assume a specially allocated software buffer is used to hold the intermediate values, and model this swapping event by causing the thread switching pipelines to issue 31 store instructions into its current pipeline, followed by 31 load instructions into the other pipeline. This is a conservative mechanism for thread switching, but any realistic mechanism will be slow, and this assumption forces us to deal with that latency.

This scheme incurs two new performance costs:

(1) Increased register conflicts — With a single (logical) register file, we run out of renaming registers only when all registers in the machine are allocated. However, now that registers are allocated separately from each register file, a

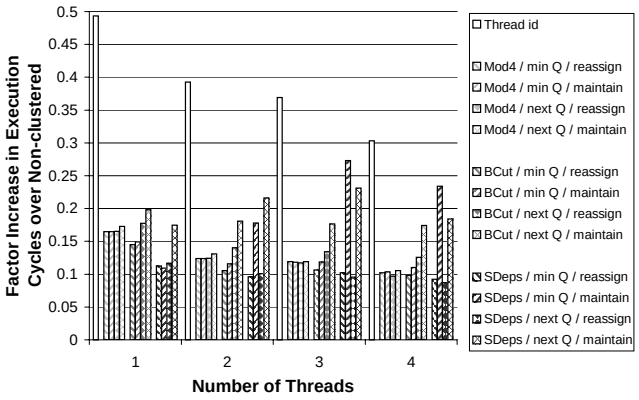

Figure 2. Factor of increase in execution cycles due to queue clustering compared to a non-clustered processor running the same number of threads. Results are shown for various queue assignment schemes as the mechanisms for choosing the next queue and response to a full queue are varied.

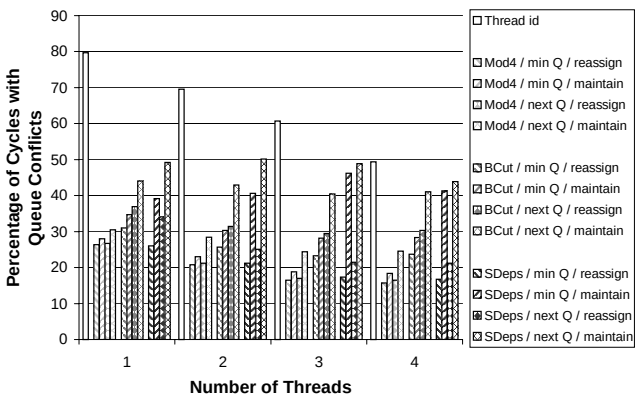

Figure 3. Percentage of cycles on which queue conflicts occur. Results are shown for various queue assignment schemes as the mechanisms for choosing the next queue and response to a full queue are varied.

register conflict occurs whenever either register file runs out of registers.

(2) Thread switch delays — When a thread switches the pipeline it fetches into, it must now explicitly copy all integer register state between the two independent register files.

5. Performance Impact of Splitting Processor Structures

This section examines the performance impact and design space for each of the architectures described in the previous section. We focus on (1) understanding the interaction of clustering with multithreading and (2) maximizing total performance by designing instruction and thread steering mechanisms which minimize the IPC loss relative to a non-clustered architecture. Results in this section assume a multithreaded workload composed of permutations of the single-threaded SPEC benchmarks.

5.1. Design of a queue clustering architecture

In the processor with partitioned instruction queues, instructions are assigned to an instruction queue immediately after they are renamed. Various queue assignment schemes have been proposed which assign instructions to queues, utilizing instruction specific information. For example, the branch-cut scheme [4] assigns instructions to the same instruction queue until a branch is encountered, at which point a new queue is chosen. Each of these schemes is extended and modified to work within a multithreaded architecture.

Our instruction steering mechanism explores three somewhat independent policy choices: When does a thread switch queues? Which queue does it switch to? How permanent is the queue assignment? For the first option, we consider a high-performing subset of previously proposed instruction steering mechanisms:

(1) **mod4** switches queues every 4 instructions [4].

(2) **branch-cut** switches queues after each branch [4].

(3) **seek dependencies** sends instructions to the queue which contains the younger parent of the instruction under consideration, unless all operands are already available [4].

(4) **thread id** is a new policy that only makes sense on a multithreaded processor. This scheme eliminates clustering-induced instruction latency penalties by always sending instructions from the same thread to the same cluster.

The choice of target queue after a switch is governed by one of two policies: The *next queue* scheme maintains a next queue for each renaming cluster (shared by all threads). When queried, it returns the id of the next queue in round-robin order. The other scheme, *fewest instructions*, always chooses the queue containing the fewest total instructions.

The last policy governs the processor's response to queue conflicts. Normally when an instruction is assigned to an instruction queue the assignment is permanent (the *queue maintain* policy). If the particular instruction queue is full, however, this approach can result in excessive queue conflicts. Thus, we also evaluate *queue reassign*, which permits instructions which cause queue conflicts to be reassigned to a different instruction queue on the cycle following the conflict (as well as following instructions).

Figure 2 shows the result of various combinations of these policies. It shows that for well-performing policies, multithreading makes the processor significantly more tolerant of cluster delays. It also makes the processor more tolerant of steering policies; policies which are not competitive at one thread can be very competitive with the most complex schemes at four threads (for example, the simple *mod4* is similar to the best *seek dependencies* with four threads).

The *seek dependencies* policy typically outperforms other approaches. However, it is highly sensitive to the queue reassignment policy. Schemes which tend to insert a large number of instructions into the same instruction queue over a short period are more likely to cause a queue conflict. When this happens, not allowing instructions to be reassigned is disastrous, and not steering instructions to the shortest queue

when possible also hurts. Recall that a poor cluster assignment stalls one thread, but a queue conflict impacts all threads. Thus, those schemes which don't necessarily optimize cluster assignment only improve with increased threading, but those that increase queue conflicts can get significantly worse. The simple *mod4* scheme naturally spreads instructions among all queues, achieving fairly consistent performance for any number of threads, regardless of the other queue policies. In fact, the very simplest scheme, *mod4 / nextQ / maintain* is quite competitive with the other schemes if there is sufficient thread-level parallelism.

Simply partitioning assignment by *thread id* works very poorly, even with sufficient threads. In this configuration, the processor takes little advantage of the diversity between threads, and load balance issues which cause queue conflicts eventually stall the whole processor.

For this configuration, performance is highly correlated with the number of queue conflicts incurred by the instruction scheduling strategies. Figure 3 shows the percentage of cycles on which an instruction queue conflict occurs. Some configurations, such as the *SDeps / next Q / maintain* configuration, result in a significant number of queue conflicts. A single-threaded processor will suffer slightly due to *load imbalance* (a queue conflict when another queue is not full). However, the multithreaded processor is very sensitive to a *stalled shared resource* (in this case, the shared pipeline stage which feeds the instruction queues). For the *thread id* policy, it only takes one thread to experience queue conflicts to stall the whole processor, which happens frequently. Additionally, with this policy no queues get to take advantage of the inherent parallelism between instructions from different threads.

5.2. Renaming Clustering

Splitting the register renaming logic reduces pressure on the renaming structures, and significantly reduces the complexity of the mapping logic (which is linear in delay and quadratic in area as a function of renaming width). However, it can lead to performance losses from both direct and indirect sources. Renaming conflicts directly stall the pipeline when more than four instructions are sent to the same renaming cluster, or when instructions are sent to a cluster which hasn't yet received the thread's most recent renaming information. Indirectly, the effectiveness of our queue clustering scheme is restricted because each renaming cluster has access to only two of the four instruction queues, leading to increased queue conflicts.

We explore two general approaches to renaming cluster assignment — *deferred* assignment and *fixed* renaming schemes. In both cases the entire fetch group uses the same renaming cluster. Some of our fixed assignment schemes define a short-term mapping of threads to renaming clusters, grouping threads according to some metric (for example, performance), and keeping that grouping for a fixed number of cycles.

Deferred assignment assigns instructions to a cluster immediately before they are renamed. For example, if one instruction group can be renamed in only one renaming cluster (because its most recent renaming information hasn't yet propagated to the other cluster) and another can be renamed in either, then the groups will be assigned such that both are renamed. When both instruction groups can be renamed in either cluster they are assigned randomly. Deferring renaming assignment not only allows instructions to bypass renaming conflicts, but also allows instructions to continue to be renamed even when the instruction queues following one of the renaming units are full.

After renaming instructions in a particular renaming cluster, a thread cannot send instructions to the second renaming cluster until that unit receives this most recently generated renaming information. However, because communication between renaming clusters requires a constant amount of time, it is trivial to count the number of cycles which have elapsed since a particular cluster was last used, ensuring that no instructions are sent to the other cluster until this delay has elapsed. This policy could be extended to handle configurations with more than two renaming clusters.

We consider the following renaming assignment schemes:

(1) **Deferred** renaming assigns instruction groups to renaming clusters immediately before they are renamed, in such a way as to minimize the number of instructions which cannot be renamed in the following cycle.

(2) **Fetch-time** assigns instructions to renaming clusters similar to the deferred scheme, but does so at fetch-time, and thus is a fixed assignment scheme. To minimize renaming conflicts, a thread is only fetched if its instructions can be assigned to the renaming cluster it last used, or if it has not been fetched for at least as many cycles as the renaming delay (a likely indication that renaming information will be up to date in both clusters when these instructions reach it). If both threads can be assigned to either cluster, they are assigned randomly.

(3) **Balance-IPC** observes thread behavior (in this case instructions committed in each cluster) and attempts to map threads to clusters so as to balance this metric across the clusters. Thread behavior is recorded for some period of time (e.g., 512 cycles), and then a mapping is created. The created mapping assigns threads to clusters so as to minimize the difference in instruction throughput in the two clusters, assuming the threads do the exact same thing they did in the measurement period. This mapping is used for the next period, during which time new information is gathered for the following period. This is also a fixed assignment scheme.

(4) **Time-based** is a fixed scheme that rotates threads through all balanced assignments of threads to clusters (in which the number of threads assigned to each cluster is as even as possible). This scheme spends only a portion of the time in the worst performing configuration, yielding average case, rather than worst case, performance.

In addition to *balance-IPC*, a number of schemes of this class were investigated which observed different attributes of the instruction stream (instructions committed, cache behavior, branch behavior etc.), including symbiosis-based schemes similar to those explored in [19]. However, we found that most schemes performed similarly. Because well-performing threads often share many characteristics (high

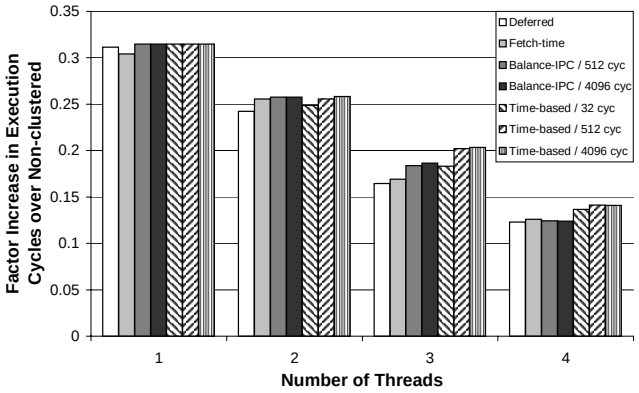

Figure 4. Factor of increase in execution cycles due to queue clustering compared to a non-clustered processor. Results are shown for various renaming cluster assignment schemes as the renaming delay is varied.

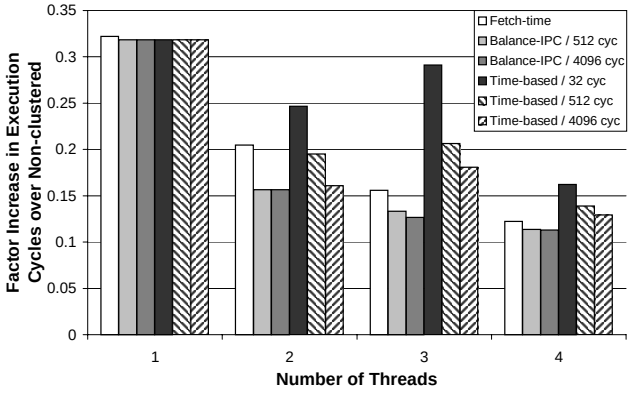

Figure 5. Factor of increase in execution cycles due to front-end clustering compared to a non-clustered processor for various pipeline assignment schemes.

IPC, low branch misprediction rate, high cache hit rate, and cause few instruction queue conflicts), adapting to any of these attributes would tend to group the same threads together. Thus, we present results only for the best performing of these schemes (balance-IPC).

Figure 4 compares the IPC from our different renaming clustering schemes as the number of executing threads is varied between one and four. The investigated schemes, shown from left to right, are: *deferred* assignment, *fetch-time* assignment, *balance-IPC* and *time-based* schemes. Multiple *balance-IPC* and *time-based* configurations are shown, assuming different cluster swapping frequencies. All configurations assume a renaming delay (the latency for mappings to be communicated to another cluster) of three cycles.

The trend observed when clustering the instruction queues is continued here — as the number of executing threads is increased, clustering-related IPC losses decrease. However, we see that none of the clustering policies has as much of an impact on performance as the amount of thread parallelism. The indirect effects of renaming clustering (restricting our ability to distribute instructions to queue clusters) exacerbates the stalled shared resource problem and plays a dominant role when multiple threads execute. Using the most aggressive renaming scheme (*deferred*) yields an increase of 14% in cycles during which queue conflicts occur for two executing threads, and 7% for three threads.

The two-thread configuration is not significantly impacted by the clustering scheme used. The single thread configuration performs poorly due to the 4-instruction single-thread throughput limit. Section 6 explores techniques to mitigate single-application performance losses.

In this configuration, threads swap clusters rarely since a non-zero renaming delay prevents a thread from being renamed in different clusters on back-to-back cycles. However, the *deferred* scheme does allow threads to switch in response to a queue conflict, showing a slight benefit over other approaches.

When more than two threads execute, the more aggressive schemes, *deferred assignment* and *fetch-time*, achieve very good performance. Both these schemes minimize the *stalled shared resource* problem, which in this case is the pipeline stage which feeds into the renaming stage.

In other experiments (not shown), we found that performance was only slightly improved by reducing the renaming delay to one cycle, despite a smaller number of renaming conflicts occurring. Renaming conflicts play a smaller performance role than queue conflicts because a renaming conflict only prevents the second group of four instructions from being renamed on a particular cycle, whereas a queue conflict may prevent any instructions from being renamed on that cycle.

5.3. Front-end clustering

Instruction throughput losses in the previous architecture are primarily due to increased susceptibility to the stalled shared resource problem when back-pressure causes the earlier fetch and decode stages to stall. If we also cluster those stages, complexity is decreased slightly (e.g., in the fetch unit), but we also make it much more difficult for a single thread to clog up the entire pipeline. Splitting the processor's eight-wide front-end (fetch, decode and other pipeline stages which precede register renaming) yields two independent, four-wide pipelines, each of which leads to a private register renaming stage, private instruction queues, and separate functional units. In this architecture, instructions cannot switch pipelines after being fetched.

We utilize modified versions of the assignment schemes described in the previous section. In this case such schemes determine which pipeline to fetch into, rather than simply which renaming cluster to use.

Figure 5 presents the performance of a number of front-end renaming assignment schemes as the number of threads is varied between one and four. Schemes shown are, from left to right: *fetch-time*, *balance-IPC*, and *time-based* swapping schemes. For this processor configuration, we assume that a longer, five-cycle renaming delay will be necessary.

As before, we see that multithreading still provides tolerance of clustering delays, even with the heavy partitioning

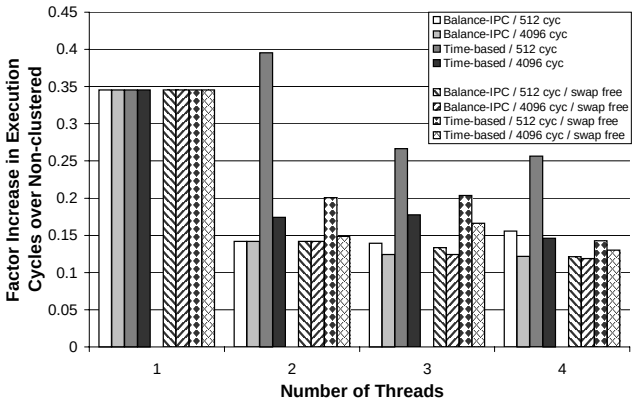

Figure 6. Factor of increase in execution cycles due to register file clustering compared to a non-clustered processor. Results are shown for *balance-IPC* and *time-based* schemes, modeling a cost to swap register files (left) and no cost (right).

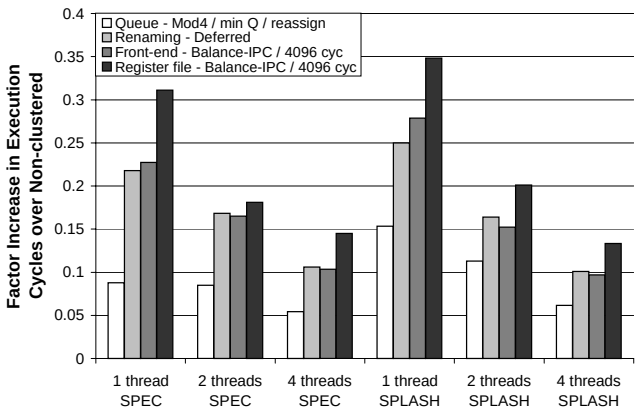

Figure 7. Average factor of increase in execution cycles for manually and automatically parallelized benchmarks for different clustering configurations. Slowdown is shown relative to a non-clustered processor with the same number of threads.

we have here.

The primary factor in the performance of these results is the frequency of switching. Less is gained in this architecture by switching, and the costs are higher. The *time-based / 32 cyc* scheme switches too often and incurs a large number of out-of-order renaming conflicts.

When threads switch pipelines less often, these performance losses are avoided. Performance is much less sensitive to queue conflicts because only the pipeline responsible for the conflict is stalled. *Balance-IPC* performs better than *time-based* both because it tends to group threads in better performing combinations, and because threads swap pipelines less often; when the new mapping agrees with the old mapping, no threads change pipelines.

In addition to simplifying control and providing an efficient mechanism for clustering register renaming, clustering the front end also opens the door to splitting further processor structures which would not have been feasible to cluster with a unified front-end, such as the register file, as will be shown in the next section.

5.4. Register File Clustering

Up to this point in this research, the register file remains one of the few remaining large shared processor structures. Normally this structure is difficult to split, both because of the difficulty in determining which register values should be stored in which smaller register file, and because of the inefficiency of partitioning the renaming registers. A processor with a clustered front-end presents a platform more suited to splitting the register file; a separate private register file is added to each pipeline, and a register conflict will stall only the pipeline in which it occurs.

Because register values are no longer directly accessible from both pipelines and must be explicitly copied between clusters (at nontrivial costs), only *balance-IPC* and *time-based* schemes are feasible, and only those which retain mappings for significant time.

Figure 6 shows the performance impact of clustering the register file as the number of threads is varied from one to four. We model two instruction assignment schemes (*time-based* and *balance-IPC*), and for each scheme show performance modeling a cost for threads to swap clusters (requiring the execution of 31 stores followed by 31 loads, as described in Section 4), and another set of results assuming threads swap clusters at no cost.

Due to the clustered processor front-end, the performance loss from splitting the register file is minimal; our best performing configuration (*balance IPC / 512 cyc*) performs typically within 2% of a processor with a unified register file. While the actual mapping of threads to clusters matters (*balance-IPC* continues to outperform *time-based*), the frequency of thread swapping plays a dominant performance role. Recall that *balance-IPC* does not cause threads to swap clusters when the newly established mapping agrees with the prior mapping, resulting in significantly fewer total swaps. Overall, we see that in a processor with a clustered front-end, it is feasible to cluster the register file as well.

6. Single Application Performance

To this point, we have concerned ourselves primarily with the performance of an aggressively clustered processor running a multiprogrammed workload; but even when clustering only instruction queues, single thread performance can be significantly reduced. While we have shown that increasing the number of executing threads is an effective technique to reduce clustering-related IPC losses, that approach is not always possible – sometimes the workload on a multithreaded processor is only composed of a single application. This section explores that scenario in more detail.

This section will explore the performance of applications parallelized by offline and online techniques when running on the the family of clustered SMT processors we have defined.

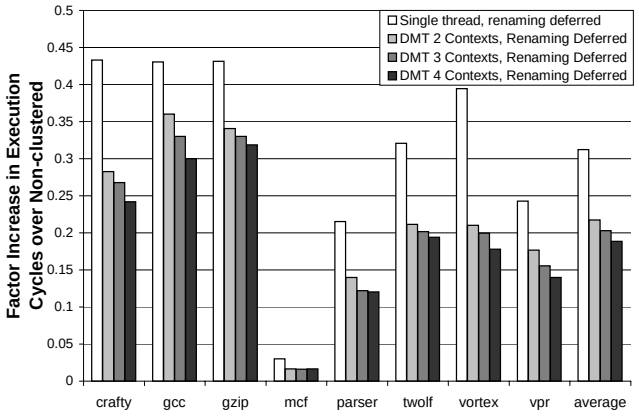

Figure 8. Factor of increase in execution cycles when applying register renaming clustering to a processor augmented with Dynamic Multithreading.

We show results (Figure 7) for five explicitly parallelized SPLASH2 benchmarks (`ocean`, `radiosity`, `raytrace`, `volrend` and `water-nsquared`) and a selection of the SPEC benchmarks (`applu`, `equake`, `mgrid` and `swim`) which have been automatically parallelized via the SUIF [9] compiler.

All benchmarks were fast forwarded over initialization code, and 500 million total instructions (or until program completion) were simulated from the point at which multiple threads began executing. Average results are shown for each group of benchmarks, corresponding to one, two and four thread configurations.

As we increase the number of threads, not only does overall performance improve, but the slowdown incurred due to clustering decreases as well. Adding a second executing thread dramatically reduces slowdown compared to more aggressive clustering schemes, and executing four threads continues this trend. Averaging over all benchmarks, the most aggressive configuration (register file clustering), when executing four threads, sees only a 13.9% loss in IPC compared to the non-clustered processor, in contrast to the 33.2% loss for a single thread. Thus, for applications that can be parallelized, the multithreaded architecture again allows the processor to retain a much higher portion of the clock rate improvements.

Current and emerging research is exploring techniques for parallelizing programs which cannot be safely parallized by a compiler. These techniques include threaded multi-path execution [26], helper threads such as speculative precomputation [6, 7, 28], and speculative multithreading [2, 21, 13].

We specifically model the IPC impact of Dynamic Multithreading (DMT) [2] on a processor implementing register renaming clustering. The results, shown in Figure 8, are shown as slowdown compared to a non-clustered DMT processor with the same total number of thread contexts, and the processor is clustered with deferred renaming clustering and the *mod4 / minQ / reassign* queue scheme.

The results indicate that such techniques have the potential to mitigate clustering related performance losses just as the more conventional multithreading techniques do. While with a single thread, we observe significant pformance losses (31%), adding a second available thread context reduces these costs (21%) compared to a non-clustered configuration. As additional thread contexts are added, these performance losses are further reduced, down to only 18% for four total thread contexts.

7. Conclusion

This paper examines the design space of a wide selection of heavily clustered, simultaneous multithreading processor architectures. The results demonstrate several key principles that impact the design of these systems.

(1) SMT processors can hide much of the latency incurred by clustering. For example, when clustering the register renaming hardware, multithreading reduces the performance cost of clustering from 30.4% down to only 12.3%. This enables more aggressive clustering approaches than might have been feasible without multithreading, enabling the processor to be designed more aggressively for low complexity and high clock rate than a non-multithreaded architecture. This architecture can sustain both higher IPC and higher clock rates than architectures that do not combine both of these techniques. (2) We show that an SMT processor in several cases enables the use of less complex instruction steering algorithms to get the same performance, reducing complexity in that way as well. (3) This paper introduces new instruction and thread steering options that are either not feasible or not possible in a conventional architecture. (4) The dominant performance phenomenon for clustered SMT processors is that of stalled shared resources (a phenomenon that is not relevant for non-multithreaded architectures). Instruction steering mechanisms need to be designed carefully. Those that either minimize these effects, or eliminate them (by allowing other threads to bypass stalled stages) are the most effective. (5) Even single-application workloads can tolerate clustering delays by utilizing the thread hardware. Running a single application using multiple threads reduces the cost of clustering from 33.2% down to 13.9%. Emerging techniques for accelerating a single thread using idle multithreading hardware can also be applied, and we demonstrate how Dynamic Multithreading [2] reduces the performance impact of renaming clustering from 31.1% down to 18.8%.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded by a Focht-Powell fellowship, a grant from Intel Corporation, and NSF grant CCR-0105743.

References

[1] A. Aggarwal and M. Franklin. An empirical study of the scalability aspects of instruction distribution algorithms for

clustered processors. In *International Symposium on Performance Analysis of Systems and Software*, Nov. 2001.

[2] H. Akkary and M. Driscoll. A dynamic multithreading processor. In *31st International Symposium on Microarchitecture*, Nov. 1998.

[3] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically managing the communication-parallelism trade-off in future clustered processors. In *30th Annual International Symposium on Computer Architecture*, June 2003.

[4] A. Baniasadi and A. Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. In *33rd International Symposium on Microarchitecture*, Dec. 2000.

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

[6] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous subordinate microthreading (ssmt). In *26th Annual International Symposium on Computer Architecture*, pages 186–195, Oct. 1999.

[7] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *28th Annual International Symposium on Computer Architecture*, pages 14–25, July 2001.

[8] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *30th International Symposium on Microarchitecture*, Dec. 1997.

[9] M. Hall, J.-A. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and M. Lam. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer*, 29(12):84–89, 1996.

[10] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *19th Annual International Symposium on Computer Architecture*, pages 136–145, May 1992.

[11] R. Kessler. The alpha 21264 microprocessor. In *IEEE Micro*, March/April 1999.

[12] V. Krishnan and J. Torrellas. A clustered approach to multithreaded processors. In *12th International Parallel Processing Symposium (IPPS)*, Mar. 1998.

[13] P. Marcuello and A. Gonzalez. A quantitative assessment of thread-level speculation techniques. In *International Parallel and Distributed Processing Symposium*, May 2000.

[14] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture: A hypertext history. *Intel Technology Journal*, 6(2), Feb. 2002.

[15] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The case for a single-chip multiprocessor. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

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

[17] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *30th International Symposium on Microarchitecture*, Dec. 1997.

[18] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. In *29th Annual International Symposium on Computer Architecture*, May 2002.

[19] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[20] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[21] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, Jan. 1998.

[22] D. Tullsen. Simulation and modeling of a simultaneous multithreaded processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[23] D. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreaded processor. In *34th International Symposium on Microarchitecture*, Dec. 2001.

[24] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[25] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[26] S. Wallace, B. Calder, and D. Tullsen. Threaded multiple path execution. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[27] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *Conference on Parallel Architectures and Compilation Techniques*, pages 49–58, June 1995.

[28] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *28th Annual International Symposium on Computer Architecture*, pages 2–13, July 2001.