Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduled, Superscalar Processors

Amirali Baniasadi
Electrical and Computer Engineering
Northwestern University
amirali@ece.northwestern.edu

Andreas Moshovos
Electrical And Computer Engineering
University of Toronto
moshovos@eecg.toronto.edu

Abstract

We investigate instruction distribution methods for quad-cluster, dynamically-scheduled superscalar processors. We study a variety of methods with different cost, performance and complexity characteristics. We investigate both non-adaptive and adaptive methods and their sensitivity both to inter-cluster communication latencies and pipeline depth. Furthermore, we develop a set of models that allow us to identify how well each method attacks issue-bandwidth and inter-cluster communication restrictions. We find that a relatively simple method that changes clusters every other three instructions offers only a 17% performance slowdown compared to a non-clustered configuration operating at the same frequency. Moreover, we show that by utilizing adaptive methods it is possible to further reduce this gap down to about 14%. Furthermore, performance appears to be more sensitive to inter-cluster communication latencies rather than to pipeline depth. The best performing method offers a slowdown of about 24% when inter-cluster communication latency is two cycle. This gap is only 20% when two additional stages are introduced in the front-end pipeline.

1 Introduction

Exploiting instruction-level parallelism via out-of-order execution facilitated rapid performance improvements during the past decade. An evolutionary path to continuing this performance growth calls for larger and wider instruction windows. The hope is that such instruction windows will expose more parallelism leading to higher concurrency and hence higher performance. Unfortunately, it is now widely believed that simply scaling the existing centralized window designs may not be possible without adversely affecting clock cycle and consequently performance. There are several reasons why including fundamental scaling limitations of centralized designs [12] and changing semiconductor technology trade-offs, e.g., [2,10] (e.g., it may not be possible to route results within a single cycle in a wide superscalar processor).

Accordingly, *clustering* has been proposed as an alternative to wide and deep organizations. In clustering, a collection of smaller windows with associated functional units is used to approximate a much wider and deeper window. Compared to a centralized organization, clustered designs trade-off scheduling flexibility for higher clock rates. Consequently, to achieve high performance we need to distribute instructions among the clusters so that clustering-induced stalls are minimized. Such stalls are primary the result of restricted intra-cluster issue bandwidth and of increased inter-cluster communication latency.

Previous work investigated various instruction distribution (or, cluster assignment) methods for dual-cluster designs [4,6,12] (see Section 6 for additional information). Moreover, the ALPHA 21264 processor already uses a dual-cluster core [9]. Building even wider and deeper windows may require additional clusters. However, whether such designs are appropriate requires close investigation of the underlying trade-offs. Accordingly, in this work we investigate instruction distribution methods for a quad-cluster, dynamically-scheduled superscalar organization. We investigate a variety of methods with various cost, complexity and performance characteristics including *adaptive* and *non-adaptive* methods. Non-adaptive methods use fixed policies that do not change during run-time, while adaptive methods may change their decisions based on past behavior. We study methods that utilize various types of information, including dependences, dataflow graph depth, instruction types and past behavior. To gain additional insight we also vary intra-cluster issue and inter-cluster communication restrictions. Finally, we investigate the sensitivity of these methods to relevant architectural parameters, i.e., inter-cluster communication latency and pipeline depth.

Some (i.e., the non-adaptive), but not all of the methods have been proposed and evaluated before in the context of dual-cluster processors. To the best of our knowledge, no other study of instruction distribution heuristics for quad-clustered, dynamically-scheduled superscalar processors has been published. Of course, there is a multitude of architectural parameters that are relevant for clustered designs and for the methods we studied. Moreover, cluster-aware compiler scheduling techniques warrant further attention. However, such an extensive investigation is not possible given the limited space available. Nevertheless, we study a variety of representative configurations varying a set of key architectural parameters.

The rest of this paper is organized as follows. In Section 2, we briefly discuss a number of trade-offs relevant to the design of instruction distribution methods. Here, we also discuss our methodology. In Section 3 we present a number of non-adaptive heuristics and evaluate their performance. In Section 4, we discuss a number of adaptive heuristics. In Section 5, we investigate the sensitivity of the better performing heuristics to increased inter-cluster communication latency and front-end pipeline stages. In Section 6 we review related work. Finally, in Section 7 we summarize our findings and offer concluding remarks. For clarity we use the term *distribution method* in

place of *instruction distribution method.* We also use *communication* instead of *inter-cluster communication*. Finally, we use the terms *centralized* and *non-clustered* architecture interchangeably.

2 Distribution Trade-offs and Methodology

In this Section, we discuss the trade-offs involved in developing instruction distribution methods. Throughout this study we assume a uniform, quad-cluster organization (the details are given later in this Section). The front-end delivers instructions which are then distributed to the four clusters via a distribution mechanism. Our focus is on this distribution mechanism. As we later show, this assignment process can dramatically impact performance. We assume that each cluster contains each own scheduler and set of functional-units. Furthermore, we assume that once an instruction is assigned to a cluster the decision is final. An alternative would be to decouple execution resources and schedulers, however, such a study is beyond the scope of this paper. Each cluster has its own set of functional units including data cache ports. Dependent instructions can issue back-to-back provided that they both reside in the same cluster. However, propagating results across clusters requires additional cycles.

Throughout this study, our goal is to maximize performance through appropriate distribution methods. To achieve maximal performance an ideal schedule is needed. However, this is a hard problem even for a centralized architecture. Accordingly, it is convenient to approach distribution as a problem of minimizing clustering-induced stalls compared to an equivalent (i.e., same overall instruction window and resources) centralized architecture. Clustering-induced stalls are either the result of limited per cluster issue bandwidth (and in general, resource distribution including functional units) or of inter-cluster communication latencies.

In contrast to a centralized configuration, each cluster is limited to only a fraction of the total issue slots per cycle (for example, each of the four clusters can issue only 2 instructions of the total of 8 per cycle). Accordingly, it is possible for an otherwise ready-to-issue instruction to get stalled in one cluster while free issue slots exist in other clusters. Moreover, since we assume that it takes additional cycles to propagate results across clusters, it is possible for an instruction to get stalled waiting for data that is currently available at another cluster. However, it is not strictly true that our mechanism should minimize such stalls. To be precise, it is only those stalls that impact the critical path through the computation that are really important. It may be possible to tolerate some stalls. Accordingly, we can categorize stalls into *benign* (those that do not affect performance compared to a centralized organization) and *harmful*. An example illustrating some of the trade-offs is given in Figure 1.

While maximum performance is desirable, the potential performance benefits of a distribution method should be weighted against its cost and complexity. Of particular concern is the size of any auxiliary structures used. For example, in Section 4, we will study a number of methods that significantly improve performance while utilizing sizeable cache-like structures.

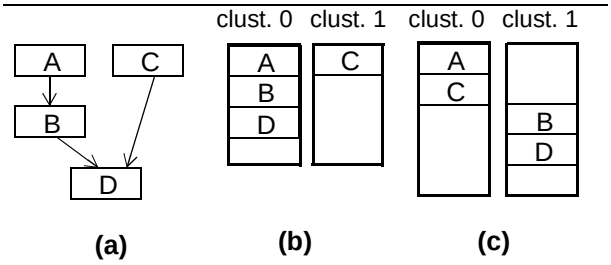

Figure 1: *Example illustrating some of the performance trade-offs in instruction distribution. We assume a dual-cluster configuration and unit latencies for all operations. Furthermore, we assume single-issue clusters. (a) Code fragment with arrows representing data dependences and boxes instructions. (b) A cluster assignment that maximizes performance. Notice that while it takes an additional cycle to propagate C's result, this does not negatively impact performance. (c) A non-optimal cluster assignment. Execution is delayed while A's result is propagated between the two clusters.*

Depending on the real-estate available on-chip, this space may be better used for other purposes (e.g., branch prediction).

Moreover, care must be taken to consider not only IPC improvements but also the potential impact on clock cycle and pipeline depth. Besides the number of steps required by the method, particular attention should also be given to the type of information used. It is desirable to use information that is readily available at the decode stage or earlier and preferably early in the clock cycle. For example, in Section 3, we will examine methods that utilize dependence information. While such information can be easily determined (e.g., via the register renaming mechanism), we may not have enough time to utilize it during the decode phase without introducing an additional stage or prolonging the clock cycle.

Before we start to describe and evaluate various methods we first discuss our methodology. We have used the SPECint'95 programs which we compiled for the MIPS-4-like architecture used by the Simplescalar simulation toolset [3]. We used GNU's *gcc* compiler (flags: -O2 -funroll-loops -finline-functions). To attain reasonable simulation times we modified the standard *train* or *test* inputs. Table 1 reports the dynamic instruction count. In the interest of space, we use the abbreviations shown under the "Ab." column.

We have modified Simplescalar's out-of-order simulator to model a variety of clustering configurations and instruction distribution methods. The base configuration is detailed in Table 2. Our base processor is capable of executing up to 8 instructions per cycle and is equipped with a 256-entry instruction window. Moreover, an 128-entry load/store scheduler (load/store queue) capable of scheduling up to four loads and stores per cycle is used to schedule load/store execution. This scheduler implements ideal memory dependence speculation [11]. Previous work has shown that memory dependence speculation is particularly important for clustered architectures. Moreover, it has been shown that it is possible to approach ideal memory dependence speculation via prediction [5,11].

Program	*Ab.*	*IC*	*Program*	*Ab.*	*IC*
126.gcc	***gcc***	1,317 M	*130.li*	***li***	207 M
129.compress	***com***	154 M	*124.m88ksim*	***m88***	196 M
099.go	***go***	134 M	*134.perl*	***per***	177 M
132.ijpeg	***ijp***	130 M	*147.vortex*	***vor***	377 M

Table 1: *Benchmark Execution Characteristics. Instruction counts ("IC" columns) are in millions.*

Default Non-Clustered Configuration			
Branch Predictor	64K GShare+64K bimodal w/ 64K selector	**Fetch Unit**	Up to 16 instr. per cycle.
Instruction Window Size	256 entries	**Load/Store Queue**	128 entries, 4 loads or stores per cycle
Issue/Decode/Commit Bandwidth	8 instructions / cycle	**Functional Unit Latencies**	same as MIPS R10000
L1 - Instruction cache	64K, 2-way SA, 32-byte blocks, 2 cycles	**L1 - Data cache**	64K, 4-way SA, 32-byte blocks, 2 cycles
Unified L2	256K, 4-way SA, 64-byte blocks, 12 cycles	**Main Memory**	Infinite, 100 cycles
Default Clustered Configuration			
Clusters	4, each 2-way issue w/ uniform distribution of functional units 64-entry windows and 32-entry load/store queues per cluster	**Inter-cluster delay**	1 cycle both for registers and store-load forwarding

Table 2: *Base configuration details. We model an 8-way aggresive, dynamically scheduled superscalar processor having a 256-entry scheduler and an 128-entry load/store queue. Also shown is the default quad-cluster configuration.*

3 Non-Adaptive Methods

We have investigated both *adaptive* and *non-adaptive* methods. Non-adaptive methods use fixed policies that do not change during run-time. For example, always selecting the cluster with fewest instructions. Adaptive methods, on the other hand, base their decisions on dynamically collected information. For example, whether the cluster assignment for a particular instruction resulted in a stall last time it was executed. In this Section, we are concerned with non-adaptive methods. Further information on adaptive methods is given in Section 4.

We have investigated a variety of non-adaptive heuristics with varying complexity and performance characteristics. Here we restrict our attention to the following representative subset: First-Fit (FF), Modulo (MOD_1 and MOD_3), Dependence-based (DEP), Slice (SLC), Branch-Cut (BC), Load-Cut (LC) and Dependence-Depth-based (DDB). The first two methods do not utilize program-related information, while the rest do. We have considered dependences, instruction types and dataflow depth as alternative sources of program-related information.

First-Fit (FF): In this method we assign instructions to the same cluster until it fills up completely. Then we move to the next cluster and do the same. The primary advantage of this method is its simplicity. A possible implementation comprises a per cluster global-AND of the occupied flags of the cluster's reservation stations (assuming an RUU-like implementation [15]) and a global current-cluster pointer. An incoming instruction is assigned to the current cluster so long there is space available (the cluster's global-AND signal is 0, i.e., there is at least one free slot available). Otherwise, the current-cluster pointer advances to the next in order cluster[1]. The impact of this method on decode/dispatch latency should be minimal as the information required is independent of the instructions themselves and can be made available early in the pipeline. While simple, this method makes no explicit attempt to minimize neither communication- nor issue-induced stalls. Nevertheless, dependent instructions tend to be close in the instruction stream. This often helps control communication-induced stalls.

Modulo Methods (MOD_n): As we will see in Section 3.1, the first-fit method fails to use issue-bandwidth efficiently. To improve issue-bandwidth utilization while keeping complexity at a minimum, we have investigated a variety of *modulo n* (MOD_n) methods. In these methods, instructions are assigned to clusters in a modulo *n* fashion where *n* is a small integer. For example, in the modulo 3 (MOD_3) method the first three instructions are assigned to cluster 0, the next three to cluster 1 and so on. Compared to FF, these methods distribute instructions more fairly among clusters resulting in a better utilization of issue-bandwidth. We have experimented with a variety of values for n and found that the optimal value differs per program. Here, we restrict our attention to MOD_1 and MOD_3. As with first-fit, the information required by modulo methods can be made available early in the pipeline. While fairly simple, MOD_3 performs surprisingly well.

Dependence-based (DEP): Neither of the methods described so far leverages program-related information. The dependence-based method uses data-dependence information in an attempt to reduce communication-induced stalls. In this method we aim to assign dependent instructions to the same cluster. This is done as follows: When decoding an instruction, we attempt to

1. Using a global "is there a free slot available?" signal per cluster makes distribution a serial process; we have to wait until the first instruction is assigned before probing for slot availability for the second instruction. To do in-parallel cluster assignment of multiple instructions we may use a population count circuit per cluster. This does not have to be complete population count circuit as the number of instructions that can be assigned per cycle is limited (i.e., decode width). Accordingly, we only care whether up to that number of slots are available per cluster.

assign it to the same cluster as its parents. If an instruction has multiple parents that are assigned to different clusters we pick the cluster holding the youngest in program order parent (i.e., closest to this instruction). (We have experimented with other alternatives and found no significant performance variations.) If the parents have long committed, we just pick the cluster with the fewest instructions. The data-dependence information required by this method can be made available via the register renaming mechanism. Depending on the particular implementation, deferring cluster assignment till after register renaming may negatively impact the clock cycle or force us to introduce additional pipeline stages.

Slice (SLC): Using the DEP method, we often find that the parents of an instruction are assigned to different clusters. This is the result of the limited, forward-dependence-based scope of the DEP method. To further reduce communication-induced stalls it would be better to assign all parents and their consuming child to the same cluster. This is the goal of the slice method. To do so, we employ the method proposed by Canal, Parcerisa and González [4]. An auxiliary, PC-indexed table (the slice table) is used to re-construct the data-flow graph on the fly. Eventually, a common tag is assigned to all instructions belonging to the same *slice*. This tag is used to assign all dependent instructions to the same cluster the next time they are encountered. If no space is available in that cluster we pick the cluster with the fewest instructions. This method reduces communication stalls since instructions within a slice will reside mostly in the same cluster. Moreover, our results show that issue bandwidth is used efficiently. However, these improvements come at the expense of an auxiliary table. Compared to DEP, the slice-table-provided tag can be made available much earlier than the register-dependence information (since the slice table is PC-indexed). We classify this technique as non-adaptive as it does not utilize explicit information about the success of past cluster assignment decisions. We assume infinite slice tables in our experiments.

Branch- and Load-Cut (BC and LC): While DEP and SLC offer superior performance they may be too complex or costly to implement depending on implementation specifics. Accordingly, we investigated methods that leverage other program-related information that can be easily extracted at run-time. In particular, we investigated methods that utilize instruction-type information. In the branch-cut method we assign consecutive instructions to the same cluster till we reach a branch instruction. The intuition behind this heuristic is that instructions within a basic-block are mostly dependent. We also investigated variations of the branch-cut method where we changed clusters only on backward branches. In doing so, we were motivated by work in thread-level speculation where loop iterations may be assigned to separate clusters for parallel execution (see Section 6). However, we didn't observe significant performance improvements. Accordingly, we restrict our attention to the general, all branch-cut method.

We also experimented with a load-cut method where instructions are assigned to the same cluster until a load is encountered. The load and the instructions that follow (till the next load) are then assigned to the next available cluster. The intuition behind this method is that loads often lead a chain of dependent instructions. Accordingly, the hope is that changing clusters upon encountering a load should force mostly dependent instructions to the same cluster, while distributing independent instructions across clusters. Whenever a sequence of adjacent loads is encountered we do not change clusters.

Dependence-Depth-based (DDB): Finally, in this method we categorize instructions based on its position (depth) in the DFG (Data Flow Graph). Only instructions currently active in the instruction window are considered in this process. If an instruction has no parents alive in the window, it belongs to depth 0. If it has only its direct parents alive it belongs to depth 1, and so on. We assign an instruction to the cluster having the least number of instructions of the same level while taking dependence information also into account (when a choice exists, we will assign to the same cluster as its closest parent). The intuition behind this method is that in a centralized configuration, instructions at the same level would probably issue around the same time (ignoring cache misses and other multi-cycle operations). Therefore by distributing them among clusters we could use the available issue bandwidth more efficiently. While this method may be fairly complex to implement, we include it as it approximates a resource-based scheduling algorithm.

3.1 Non-Adaptive Method Performance

In this Section, we present our analysis of the non-adaptive methods. We approach each method from two different perspectives. First we approach each method as an improvement over the most simple non-adaptive method (FF). Ultimately however, clustering is viable only if it results in a sufficiently higher operating frequency compared to a non-clustered implementation. For this reason, we also compare each method with a non-clustered architecture with the same overall resources.

While performance is our ultimate metric, it is desirable to get additional insight on how each method attacks issue-bandwidth restrictions and inter-cluster communication delays. To do so, we use a two tiered approach. First we report the fraction of instructions that are delayed as a result of communication or of issue bandwidth limitations. However, the two performance degrading factors interact with each other making it difficult to isolate their impact. Accordingly, we also study each performance degrading factor independently (more on this later on).

Figure 2(a) reports relative performance for DEP, BC, LC, MOD_1, SLC, DDB and MOD_3 from left to right. The base configuration is a clustered architecture utilizing the FF method. We can see that on the average, DEP performs the worst among all heuristics (excluding FF of course). With this method instructions find that their two parents are assigned to different clusters. Also, this method tends to assign too many dependent instructions to the same cluster. The first phenomenon results in inter-cluster communication induced delays, while the second phenomenon results in under-utilized issue bandwidth. As expected, (with the exception of *go*) the SLC method improves performance over DEP by placing all dependent instructions in the same cluster while distributing unrelated slices across clusters. The instruction-type-based heuristics LC and BC offer competitive and some times better performance even though they do not require an auxiliary table. The DDB method also performs similarly for most benchmarks. Further improve-

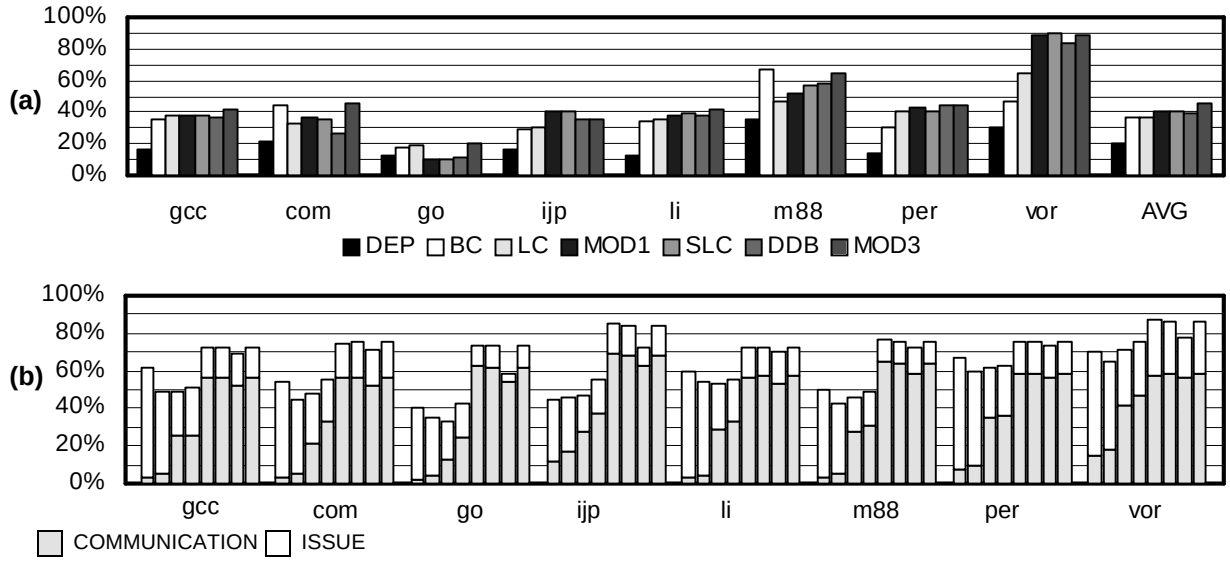

Figure 2: ***(a)*** *Performance of non-adaptive heuristics over a base configuration utilizing the first-fit (FF) method. Here we approach each method as an improvement over the simplest method we studied. Higher is better.* ***(b)*** *Fractions of committed instructions that are stalled as the result of inter-cluster communication (lower part) or issue-bandwidth restrictions (upper part). The following methods are reported: FF, DEP, BC, LC, MOD_1, SLC, DDB and MOD_3 from left to right per benchmark (same order as in part (a) with the addition of FF).*

ments may be possible by utilizing better instruction latency estimates (currently DDB assumes unit latencies for all instructions). Finally, MOD_3 performs the best. It offers a 45.6% improvement over FF. Apparently, this method strikes a good balance in assigning some dependent and some independent instructions to the same cluster. *Go* seems to benefit less from the various methods compared to the other benchmarks. This is mostly due to the relatively low branch prediction accuracy for this benchmark which results to relative small number of simultaneously active instructions. Consequently there isn't much parallelism and little room for improving performance over the simple FF method. Branch prediction accuracy is also mostly responsible for the higher performance benefits observed for vortex and to a lesser extend for m88ksim. In these programs, the vast majority of reservation stations are occupied. Moreover, these programs exhibit relatively high parallelism. Consequently, there is much to be gained by carefully distributing instructions across clusters. Moreover, these programs tend to be more tolerant to inter-cluster communication overhead (parallelism helps to tolerate these delays).

Figure 2(b) reports the fraction of committed instructions that are delayed waiting for a result from a different cluster (lower bar) or because issue-bandwidth was unavailable (upper bar). Whenever an instruction is delayed both due to inter-cluster communication and issue-bandwidth limitations we assign it to the inter-cluster communication delayed category. In general, performance and the fraction of instructions that are delayed are not correlated. However, in most cases, the best a method is, the higher the fraction of instructions that are delayed. This is the result of higher concurrency. (When distribution is not good, very few instructions are executing at any given point, resulting in very few instructions that are ready, or that would be ready if they had immediate access to the results in other clusters.) An observation can also be made about the relative fractions of instructions delayed due to communication or issued-bandwidth and performance. For the worse performing methods (FF and DEP), most instructions are delayed due to insufficient issue-bandwidth (upper bar). As we distribute instructions to better utilize issue-bandwidth, communication delays start to become more common (lower bar).

In a realistic clustered configuration, issue-bandwidth restrictions and communication delays interact making it difficult to draw conclusions. Accordingly, we introduce four machine models: **NI-NC**, **I-NC**, **NI-C** and **I-C**. In this notation, **I** indicates that the model includes per cluster issue bandwidth restrictions, while **C** indicates that communication delays are incurred. The inverse notation, **NI** and **NC**, indicates that the model does not include per cluster issue-bandwidth restrictions or inter-cluster communication delays respectively. The **NI-NC** model corresponds to a non-clustered architecture while the **I-C** model corresponds to a realistic, clustered architecture. The two other models do not correspond to realistic architectures. However, they provide additional insight on the effectiveness of each method. The **NI-C** model shows how well we could have done if no issue-bandwidth restrictions were applicable (total issue bandwidth is still limited to 8 instructions per cycle, however, these instructions can come from any cluster, possibly all from the same one). Similarly, the **I-NC** model shows how well the heuristic performs in attacking issue bandwidth restrictions (no communication stalls possible).

Figure 3 reports performance improvements over the base clustered configuration that uses the FF method. Due to space limitations we restrict our attention to FF, MOD_3, the instruction-type-based BC and the dependence-based SLC. As

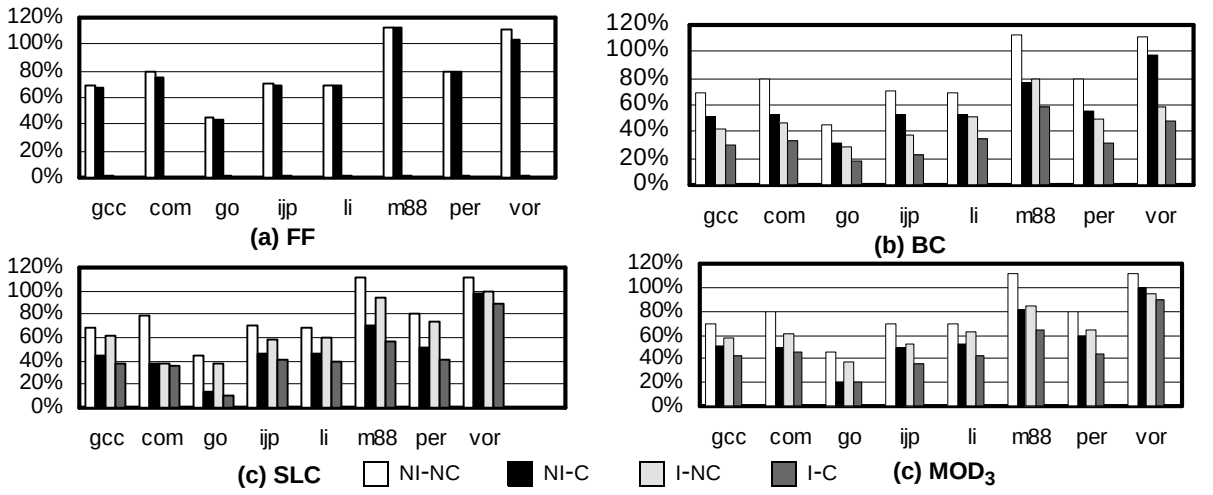

Figure 3: *How well some of the non-adaptive methods attack issue-bandwidth and communication restrictions. Four models are simulated per method. The four models are derived by selectively modeling issue-bandwidth and communication restrictions. The models are labeled with an X-Y notation, where X is either I or NI and Y is either C or NC. I indicates that issue bandwidth restrictions are imposed, while NI that they are not. Similarly, C and NC indicate that inter-cluster communication delays are modeled or that they are not respectively. NI-NC corresponds to a non-clustered architecture, while I-C corresponds to a realistic clustered one. Relative performance is reported over the base clustered configuration that utilizes the FF (first-fit) method. Higher is better.*

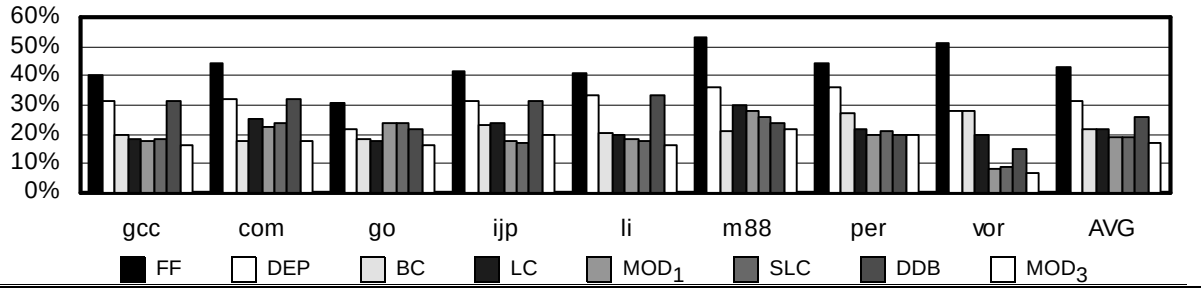

Figure 4: *Relative performance of non-adaptive method over a non-clustered organization assuming the same clock rate. Reported are slowdowns (lower is better). These slowdowns can serve as bounds on how much faster a clustered implementation's clock rate has to be over an non-clustered implementation.*

expected, the FF method (part (a)) does not perform well compared to the non-clustered architecture (NI-NC) (there are no I-C bars here since I-C with FF is the base case). In the best case of *go*, the difference is about 45%, while, it grows as large as approximately 110% for *m88ksim*. The two models NI-C and I-NC reveal that much of this performance loss is the result of issue-bandwidth distribution. When issue bandwidth is not restricted (NI-C), performance is very close to that of the non-clustered architecture (NI-NC). However, when issue is restricted and even without any communication delays (I-NC), performance drops rapidly and is very close to the realistic clustered architecture (I-C). While, the FF method is somewhat sensitive to communication delays, it is primarily crippled by inefficient use of issue-bandwidth. Issue-bandwidth restrictions seem to be more important than communication delays for BC also (part (b)). With the exception of *m88ksim*, BC performs better under NI-C than under I-NC. This trend is reversed for most benchmarks for both SLC and MOD_3 (parts (c) and (d)). As we have seen in Figure 2, these methods perform much better than either FF or BC. This result suggests that once we begin using issue bandwidth more effectively, then inter-cluster communication latencies become more important. Interestingly, the differences between I-NC and NC-I are smaller for MOD_3 for most benchmarks. This result supports our previous observation that MOD_3 strikes a better balance in attacking issue-bandwidth and communication restrictions. Notably, I-C and NI-C perform almost identical in *go*, suggesting that in this benchmark it is communication that is most important. This can be explained by the relatively low branch prediction accuracy and the resulting low instruction level parallelism.

Ultimately, a clustered architecture may be viable only if it offers better performance compared to a centralized one. For the methods we studied this can only be the result of higher operating frequency. It is desirable to know how much faster

the clock rate of the clustered architecture has to be (vs. the centralized architecture's clock rate) to result in higher performance. Accordingly, we report performance slowdowns compared to a non-clustered architecture assuming the same clock frequency. These slowdowns can serve as bounds on how much faster the clock cycle of the clustered implementation must be. The results are shown in Figure 4. MOD_3, the best non-adaptive method, is 17% slower than the centralized configuration. Notice that some minor differences in the trends exhibited compared to Figure 2, are the result of using a different base configuration (in Figure 2, we used the FF-based clustered configuration as our base).

In this Section, we have discussed and evaluated the performance of various non-adaptive heuristics. We have found that it is possible to significantly improve performance over the simplistic first-fit method. However, we have also found that there is still a sizeable gap in performance (17% on the average for the best performing method) compared to a centralized architecture operating at the same frequency. In the next section we propose methods that aim at reducing this performance gap.

4 Adaptive Methods

In this section, we present and evaluate a number of adaptive methods. The intuition behind these methods is that programs tend to exhibit non-random behavior. Accordingly, it may be possible to learn and avoid inefficient cluster assignments. We have investigated two classes of adaptive techniques. The first class is based on voting, while the second attempts to improve over the fixed modulo techniques we described in the previous section.

Voting-based Methods (CNT-X): The idea behind these methods is to identify problematic instruction assignments and try to avoid them the next time the same instructions are encountered. For example, these methods can improve instruction distribution whenever a program follows the same path repeatedly. In these methods we start with an underlying non-adaptive technique. Upon executing an instruction we record information about the success or failure of the current cluster assignment in a *Cluster Prediction Table* (CPT). We experimented with PC-indexed CPTs so that they can indexed early the pipeline. A CPT entry contains four 2-bit up/down saturating counters one per cluster. The counters indicate how appropriate a cluster might be for the matching instruction, with 11 being the best and 00 the least. Initially, all counters are set to 01, indicating that all clusters are equally appropriate. As soon an instruction becomes ready we update the corresponding counter in the CPT. If the instruction can issue immediately, we increment the counter, otherwise we decrement it[2]. The next time the same instruction is encountered, the CPT is accessed in parallel with the non-adaptive method. The instruction is then assigned to the cluster with the highest counter value (most appropriate based on past experience). If there are more than one qualifying clusters, we use either the non-adaptive method's recommendation (so long as it is one the clusters with the highest counter values) or choose the cluster with less instructions.

As described, the *voting-based* method reacts only to issue-bandwidth-induced stalls. We used these stalls as they can easily identified locally at each reservation station (ready signal vs. allowed to issue). In our simulation environment it is straightforward to also detect scenarios where inter-cluster communication is at fault. However, the specifics of a realistic implementation are beyond the scope of this paper. Accordingly, we restrict our attention to using only issue-bandwidth related stalls for our adaptive methods.

Adaptive-Modulo (MOD_a): As we have seen in Section 3, MOD_3 performed best among the non-adaptive techniques. We have also noted that the best modulo value varied per benchmark, with 3 being a good enough comprise across all benchmarks we studied. Motivated by these observations we have developed the adaptive-modulo method. In this method, we start with an initial modulo value of 3. However, as execution progresses we keep statistics on how often instructions are stalled as the result of insufficient issue-bandwidth. After a pre-specified number of instructions have executed (1 million in our experiments) we try a different modulo value (e.g., increase to 4). If the new modulo value results in fewer instructions being stalled, we continue changing the modulo value (e.g., move to 5). Otherwise, we alter our direction of change (e.g., decrement as opposed to increment). Using this policy, the modulo value is dynamically adjusted to one that offers better performance. As described, this policy can get stuck to a local maximum since it relies on comparisons between adjacent values. Accordingly, we have also tried a different policy where we sweep over a pre-specified range of modulo values (i.e., 1 to 16) before deciding on the best one (this scan is repeated at regular intervals, i.e., 100M instructions). However, we did not find significant performance improvements.

The primary advantage of this method is that it offers some adaptability without requiring many additional resources. A similar method was proposed for selecting an appropriate history-depth for branch prediction [8].

4.1 Adaptive Method Performance

We report results assuming infinite cluster prediction tables. We have also experimented with finite prediction structures and found that 16K-entry non-tagged, counter-based prediction tables perform very close and sometimes better than the infinite table (better accuracy is possible via constructive interference). As with the non-adaptive techniques, we first compare their performance using the most simple method (FF) as our base. Moreover, we report a breakdown of stalled instructions and use our four models (presented in Section 3.1) to isolate issue-bandwidth and communication related stalls. Finally, we compared with a non-clustered architecture assuming the same clock rate.

We have experimented with various voting-based methods. Here we restrict our attention to voting-based extensions to MOD_1[3], Branch-Cut (BC) and Slice (SLC). We also study the adaptive-modulo technique. Figure 5(a) shows relative performance improvements over the FF-based clustered architecture.

2. Actually, updates are done at commit time. To do so, a bit is kept in the reservation station. This bit is set when the reservation station's ready signal is set, but the ready-select logic does not allow the instruction to execute. Upon commit, the corresponding CPT entry is updated accordingly.

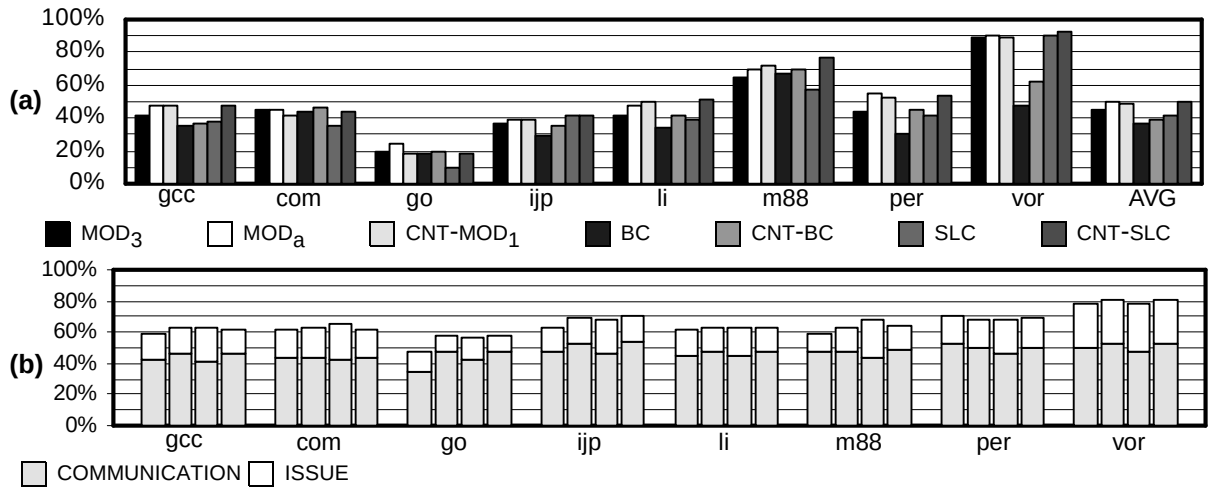

Figure 5: ***(a)*** *Adaptive Method Performance. Performance improvements are over the clustered architecture using the FF method. For ease of comparison we also include the corresponding non-adaptive methods. A CNT-X notation indicates a counter-based extension of the X non-adaptive method. Higher is better.* ***(b)*** *Fractions of committed instructions that are stalled as the result of inter-cluster communication (lower part) or issue-bandwidth restrictions (upper part). The following methods are reported: MOD_a, $CNT\text{-}MOD_1$, CNT-BC and CNT-SLC from left to right per benchmark (same order as in part (a) excluding the non-adaptive methods).*

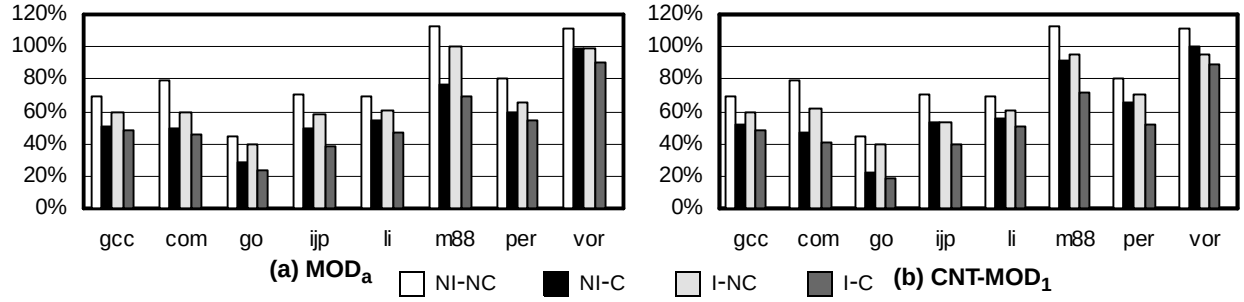

Figure 6: *How well some of the adaptive methods attack issue-bandwidth and communication restrictions. Four models are simulated per method. See Figure 3 for an explanation of the four models. Higher is better.*

For ease of comparison, the relevant non-adaptive methods are also included (repeated from Figure 2). As expected, for most cases the adaptive-techniques improve performance over the underlying non-adaptive method. On the average, the performance improvements over FF are approximately 50%, 49%, 39% and 50% for MOD_a, $CNT\text{-}MOD_1$, CNT-BC, and CNT-SLC respectively. The performance improvements over FF for MOD_3, BC and SLC were 46%, 37% and 41% respectively. The best performing method is CNT-SLC. However, $CNT\text{-}MOD_1$ and MOD_a offer very similar performance improvements. Recall, that CNT-SLC requires both a slice table and cluster prediction table. In contrast, $CNT\text{-}MOD_1$ requires only a cluster prediction table. Finally, MOD_a has minimal space requirements.

Figure 5(b) shows a breakdown of delayed instructions for the adaptive methods. The lower part of each bar reports the fraction of committed instructions that were delayed due to inter-cluster communication. The upper part shows the fraction of committed instructions delayed due to issue-bandwidth unavailability. Again, it appears that a better a method performs the more instructions are stalled. However, the differences among the various methods are small. This further justifies using the four models of issue-bandwidth and communication (see Section 3.1) to determine how sensitive each method is to each of these restrictions. The results are shown in Figure 6 where the base case is the clustered FF-based configuration. We restrict our attention to MOD_a and $CNT\text{-}MOD_1$. The general trends with respect to issue-bandwidth and communication restrictions have not changed by much. However, the gap between NI-NC and the other models has been reduced and so have, for the most part, the differences between NI-C and I-NC.

Finally, in Figure 7 we report the relative performance of our adaptive methods over a non-clustered architecture with the same clock rate. The voting-based methods perform very simi-

3. We did not observe a significant difference compared to counter-based extensions to MOD_3.

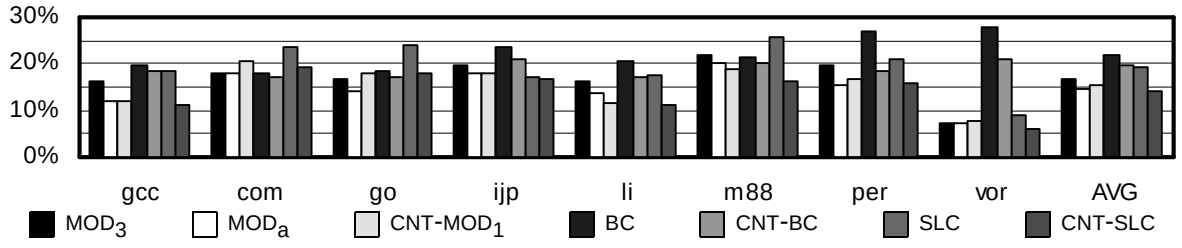

Figure 7: *Performance of the adaptive methods over a non-clustered architecture with the same clock rate. Shown are slowdowns, accordingly, lower is better.*

larly, with MOD_a offering competitive performance. CNT-MOD_1 has narrowed the gap down to 15.2%, CNT-SLC to 14.1% while MOD_a to 14.6%. In absolute terms, the improvements appear relatively minor. However, they are sizeable when compared to the original gap between the best non-adaptive method and the centralized architecture (about 3% off the maximum possible of 17%).

In this section, we have shown that it is possible to further improve performance using adaptive techniques. The best performing methods where the adaptive-modulo and the counter-based extension to MOD_1.

5 Sensitivity Analysis

In this section, we investigate the sensitivity of some methods to key architectural parameters. In Section 5.1, we vary inter-cluster communication latency, while in Section 5.2, we investigate how tolerant our mechanisms are to increases in the front-end pipeline depth. Finally, in Section 5.3, we we study configurations with 4-way, 2-way and single-issue clusters.

5.1 Inter-Cluster Communication Latency

Figure 8 reports performance when the inter-cluster communication delay is increased to two cycles. We report slowdowns over the default centralized configuration operating at the same frequency. No additional communication delays are imposed for the base configuration. As expected, the performance gap increases. CNT-SLC is the best performing method being about 24% slower than the base. MOD_3 remains the best non-adaptive method. While MOD_a still improves over the non-adaptive methods, the other, voting-based adaptive methods now perform significantly better.

5.2 Front-End Latency

We also take a look at how increasing the number of front-end pipeline stages impacts some of the best performing methods. As we discussed in Section 2, depending on the specifics of the pipeline, the information utilized and the steps required by a distribution method, we might be forced to introduce additional pipeline stages. Figure 9 reports how performance varies for one (part (a)) or two (part (b)) additional decode stages. We restrict our attention to BC, CNT-BC, SLC, and CNT-SLC. We choose these methods since they utilize program-information and/or auxiliary tables. Consequently, they are more likely to impact the depth of the front-end pipeline. We report slowdowns with respect to the default centralized configuration that does not include any additional decode stages. Inter-cluster communication latency is one cycle. Overall, the performance gap has increased. However, the relative trends do not change by much. The adaptive methods still perform better than the non-adaptive ones, with CNT-SLC being the best one.

5.3 Issue-Bandwidth

Finally, we experiment with three quad-cluster configurations: One made up of 4-way clusters, one with 2-way clusters and another with single-issue clusters. The total issue bandwidth is 16, 8 and 4 respectively. In all cases, we assume 2 cycles for inter-cluster communication. The results are shown in Figure 10, parts (a) through (c). For ease of comparison we use the 4-way centralized configuration as our base case. Numbers greater than 1 represent speedup while numbers lower than 1 represent slowdown.

As expected, the higher the issue bandwidth of each cluster the higher the performance. For the single-issue cluster configuration of part (c) all distribution methods perform similarly with the exception of BC and CNT-BC. Notably, CNT-BC performs worse that its non-adaptive counterpart BC. This is because rarely free issue-slots exist in other clusters or because selecting an alternate cluster introduces additional communication delays.

As we move to dual-issue and quad-issue clusters (parts (b) and (a)) the BC- and SLC-based methods perform better than MODa and MOD_3. CNT-MOD_1, however, remains competitive. SLC performs poorly for the 4-way cluster configuration. Recall that SLC will spread “unrelated” slices accross clusters. However, since it does not consider memory dependences, it often assigns dependent load-stores to different clusters. This results in increased store-load forwarding delays. Given that clusters are 4-way issue, we are better off assigning multiple slices to the same cluster rather than distributing them in a round-robin fashion. CNT-SLC detects innefficient cluster assignments and improves upon them. Notably, BC perfoms very close to both CNT-SLC and CNT-BC. This makes BC an attractive choice for this configuration. Recall, that the cost of BC is low compared to both CNT-BC and CNT-SLC.

6 Related Work

A plethora of studies have investigated partitioning as a way of scaling over existing, centralized dynamically-scheduled superscalar architectures. A class of methods aims at extracting parallelism by making non-continuous or large prediction-based steps in the dynamic instruction stream, e.g., [1,7,13,14,16]. Here we restrict our attention to works that

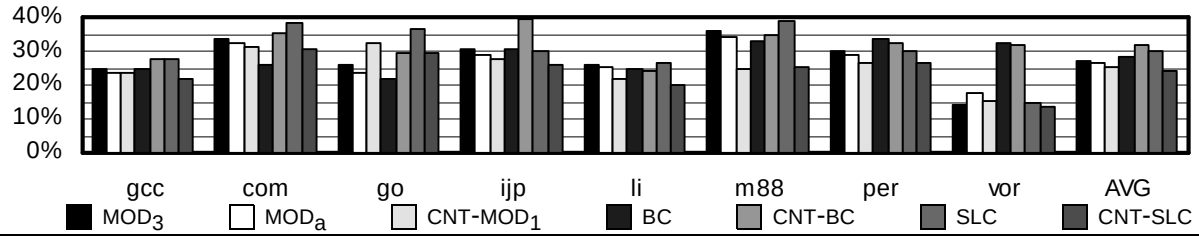

Figure 8: *Performance of some methods when inter-cluster communication latency is increased to 2 cycles. The default centralized configuration without (no communication delays) is the base. Lower is better.*

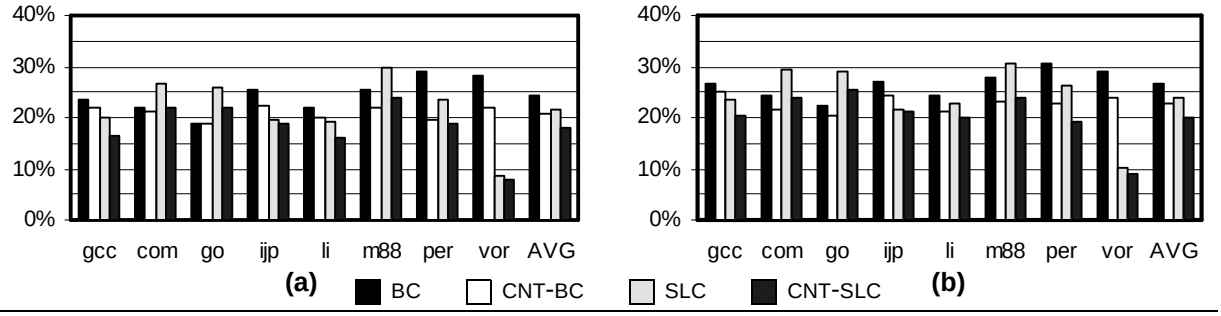

Figure 9: *Performance of some methods with deeper front-end pipelines. We include methods that utilize instruction-type and dependence information. Slowdowns are reported over the default non-clustered architecture without any additional front-end stages. Part (a) introduces one more decode stage while part (b) two.*

investigated partitioning a traditional architecture.

Palacharla, Jouppi and Smith studied the delay characteristics of key processor structures [12]. They demonstrated that it will not be possible to naively scale existing designs without adverse effects on clock cycle. They proposed using clustering as a solution and studied various non-traditional scheduling mechanisms for dual-clustered architectures (e.g., FIFO-based schedulers) and also used dependences to optimize cluster assignment. Due to the limited space available, an investigation of these alternative scheduler organizations is beyond the scope of this paper.

Farkas, Chow, Jouppi and Vransevic proposed and studied a dual-clustered architecture along with a cluster-aware static scheduling technique [6]. Canal, Parcerisa and González studied a variety of non-adaptive instruction distribution methods also for a non-uniform dual-clustered architecture [4]. They also proposed the slice-based method and explained how slice information can be extracted dynamically. Finally, the ALPHA 21264 already employs a dual-cluster micro-architecture [9].

7 Conclusion

Clustering provides a potentially viable path for wider and deeper instruction windows and higher operating frequencies. In this work, we have studied a variety of instruction distribution methods for quad-cluster processors. We studied non-adaptive methods and adaptive techniques with varying complexity and cost requirements. These methods utilized various types of information, including instruction-type, dependences and past history to better distribute instructions across clusters.

We have found that a relatively simple method, MOD_3 offers competitive performance. It was within 17% of a non-clustered organization operating at the same frequency. Moreover, we have seen that it is possible to reduce this gap down to about 14% via a counter-based prediction scheme. While in absolute terms this is a minor improvement, it does represent a sizeable reduction in relative terms (as compared to the 17% gap with MOD_3). We have also investigated the sensitivity of our methods to inter-cluster communication latency and front-end pipeline depth. We found that performance is much more sensitive to inter-cluster communication for the better performing methods. The performance gap for the best performing method increased to 24% when inter-cluster communication latency was increased two cycles. In contract, even when two additional front-end pipeline stages were introduced, this gap was only 20%.

While we studied a reasonable set of configurations and methods, there is still a plethora of design points and possible other methods that warrant further study. There are multiple directions for further experimentation, including non-uniform cluster organizations, restrictions on inter-cluster communication bandwidth, the effect of previously proposed compiler optimizations [6] and alternative scheduler designs such as those appearing in [12]. Of particular interest are organizations where execution clusters (i.e., functional units, register files and cache ports) and schedulers are decoupled. In such a design, an instruction is first assigned to a scheduler, and then, based on input operand availability is sent to the appropriate execution cluster.

Acknowledgements

We thank the anonymous reviewers and Eric Sprangle for their insightful suggestions. We also thank Scott

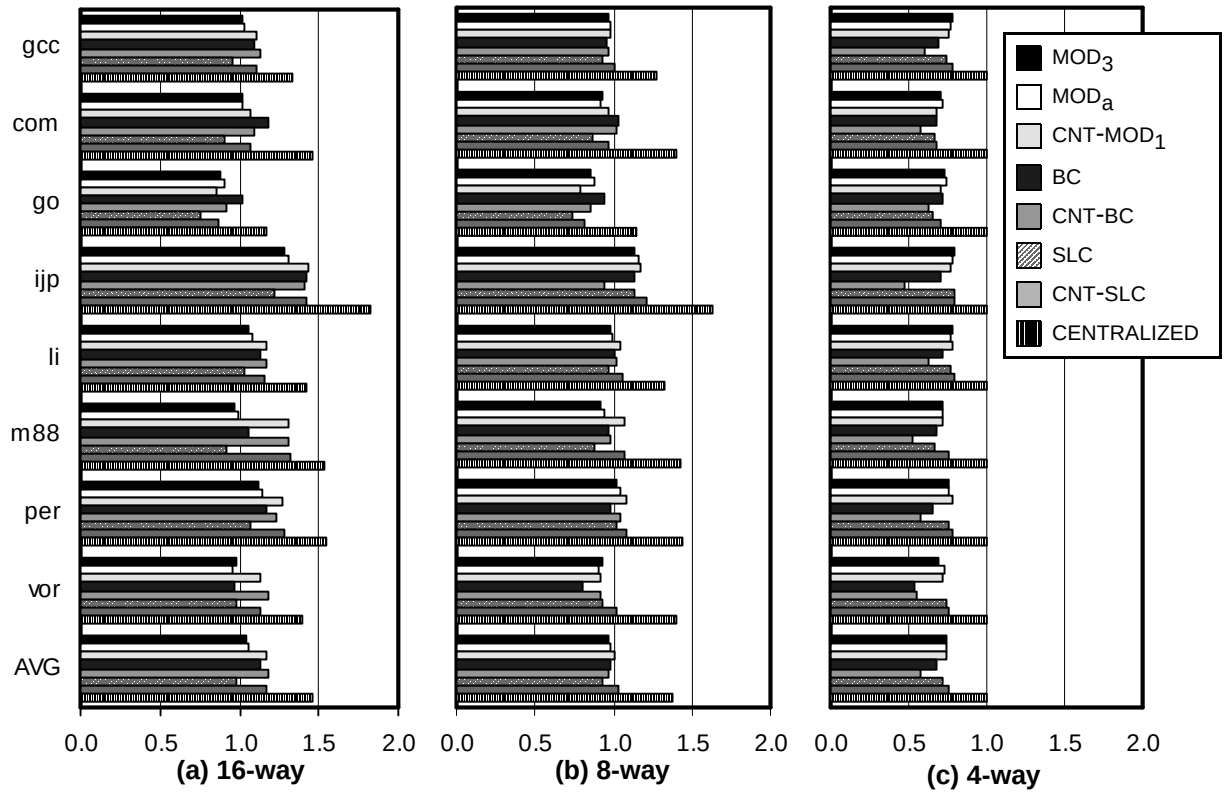

Figure 10: *Performance as a function of total issue bandwidth. All configurations use four clusters. (a) 4-way clusters (16-way total), (b) 2-way clusters (8-way total), and (c) single-issue clusters (4-way total). Performance is normalized to a 4-way centralized configuration. All clustered configurations incur 2 cycles for inter-cluster communication. The base, centralized configuration does not incur any communication delays. Higher is better.*

Breach for early discussions that lead to this study. This research was supported by an NSF Career Award.

References

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

[2] M. T. Bohr. Interconnect scaling - the real limiter to high performance ULSI. *International Electron Devices Meeting Technical Digest*, 1995.

[3] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report Computer Sciences Tech. Report #1342, University of Wisconsin-Madison, June 1997.

[4] R. Canal, J. M. Parcerisa, and A. Gonzalez. Dynamic Cluster Assignment Mechanisms. In *Proc. High Performance Architecture 6*, Jan. 2000.

[5] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proc. International Symposium on Computer Architecture-25*, June 1998.

[6] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing Cycle Time Through Partitioning. In *Proc. Annual International Symposium on Microarchitecture-30*, Dec. 1997.

[7] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proc. Symposium on Architectural Support for Languages and Operating Systems VIII* , Oct. 1998.

[8] T. Juan, S. Sanjeevan, and J. J. Navarro. Dynamic history-length fitting: A third level of adaptivity for branch prediction. In *Proc. 25th Annual International Symposium on Computer Architecture*, pages 155–166, June-July 1998.

[9] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 architecture. In *Proc. of International Conference on Computer Design*, Dec. 1998.

[10] D. Matzke. Will Physical Scalability Sabotaze Performance Gains? . In *IEEE Computer, 30(9)*, Sept. 1997.

[11] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic speculation and synchronization of data dependences. In *Proc. International Symposium on Computer Architecture-24*, June 1997.

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

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

[14] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. International Symposium on Computer Architecture-22*, June 1995.

[15] G. S. Sohi and S. Vajapeyam. Instruction issue logic for high-performance, interruptible pipelined processors. In *Proc. 14th Annual International Symposium on Computer Architecture*, pages 27–34, Pittsburgh, PA, June 1987.

[16] J. G. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proc. High Performance Computer Architecture-4*, Jan. 1998.