Improving Dynamic Cluster Assignment for Clustered Trace Cache Processors

Ravi Bhargava and Lizy K. John
Electrical and Computer Engineering Department
The University of Texas at Austin
{ravib,ljohn}@ece.utexas.edu

Abstract

This work examines dynamic cluster assignment for a clustered trace cache processor (CTCP). Previously proposed cluster assignment techniques run into unique problems as issue width and cluster count increase. Realistic design conditions, such as variable data forwarding latencies between clusters and a heavily partitioned instruction window, increase the degree of difficulty for effective cluster assignment.

In this work, the trace cache and fill unit are used to perform dynamic cluster assignment. The retire-time fill unit analysis is aided by a dynamic profiling mechanism embedded within the trace cache. This mechanism provides information about inter-trace data dependencies, an element absent in previous retire-time CTCP cluster assignment work. The strategy proposed in this work leads to more intra-cluster data forwarding and shorter data forwarding distances. In addition, performing cluster assignment at retire time reduces issue-time complexity and eliminates early pipeline stages. This increases overall performance for integer programs by 11.5% over our base CTCP architecture. This speedup is significantly higher than a previously proposed retire-time CTCP assignment strategy. Dynamic cluster assignment is also evaluated for several alternate cluster designs as well as for media benchmarks.

1. Introduction

A clustered microarchitecture design allows for wide instruction execution while reducing the amount of complexity and long-latency communication [4, 5, 6, 8, 12, 20]. The execution resources are partitioned into smaller units. Within a cluster, communication is fast, but inter-cluster communication is more costly. Therefore, the key to high performance on a clustered microarchitecture is assigning instructions to clusters in a way that limits data communication between clusters.

During cluster assignment, an instruction is designated to execute on a particular cluster. This assignment process can be accomplished statically, dynamically at issue time, or dynamically at retire time. Static cluster assignment is traditionally done by a compiler or assembly programmer. Studies that have compared static and dynamic assignment conclude that dynamic assignment results in higher performance [4, 16].

Dynamic issue-time cluster assignment occurs after instructions are fetched and decoded. In recent literature, the prevailing philosophy is to assign instructions to a cluster based on data dependencies and workload balance [4, 12, 16, 20]. The precise methodology varies based on the underlying architecture and execution cluster characteristics.

Typical issue-time cluster assignment strategies do not scale well. Dependency analysis is an inherently serial process that must be performed in parallel on all fetched instructions. Therefore, increasing the width of the microarchitecture further delays and frustrates this dependency analysis (also noted by Zyuban et al. [20]). Accomplishing even a simple steering algorithm requires additional stages early in the instruction pipeline.

In this work, the clustered execution architecture is combined with an instruction trace cache, resulting in a clustered trace cache processor (CTCP). A CTCP achieves a very wide instruction fetch bandwidth using the trace cache to fetch past multiple branches in a low-latency and high-bandwidth manner [14, 15, 18].

The CTCP environment facilitates the use of retire-time cluster assignment, which addresses many of the problems associated with issue-time cluster assignment. Cluster assignment is accomplished at retire time by physically (but not logically) reordering instructions within a trace cache line so that they are issued directly to the desired cluster. The issue-time dynamic cluster assignment logic and steering network can therefore be removed entirely, eliminating critical latency from the front-end of the pipeline.

Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)
1063-6897/03 $17.00 © 2003 IEEE

Friendly et al. present a retire-time cluster assignment strategy for a CTCP based on intra-trace data dependency analysis [7]. The trace cache fill unit is capable of performing advanced analysis, since the latency at retire time is more tolerable and less critical to performance [7, 9]. The shortcoming of this strategy is a loss of dynamic information. Inter-trace dependencies and workload balance information are not available at instruction retirement.

In this work, we increase the performance of a wide issue CTCP using a feedback-directed, retire-time (FDRT) cluster assignment strategy. Extra fields are added to the trace cache to accumulate inter-trace dependency history. The fill unit combines this information with intra-trace dependency analysis to determine cluster assignments.

This novel strategy increases the intra-cluster data forwarding by 55% while decreasing the average data forwarding distance by 40% over our baseline four-cluster, 16-way CTCP. This leads to an 11.5% improvement in performance over our base architecture compared to a 3.1% improvement for Friendly's method.

In the remainder of this paper, the architecture of our baseline clustered trace cache processor and cluster assignment schemes are presented. Section 3 contains a characterization of the dynamic instruction stream. Section 4 illustrates our feedback-directed, retire-time (FDRT) cluster assignment strategy. Section 5 presents an analysis of the overall performance improvement and additional insights. A summary concludes the paper.

2. A Clustered Trace Cache Processor

A clustered microarchitecture is designed to reduce the performance bottlenecks that result from wide-issue complexity [12]. Structures within a cluster are small and data forwarding delays are reduced as long as communication takes place within the cluster.

The target microarchitecture in this work is composed of four, four-way clusters. Four-wide, out-of-order execution engines have proven manageable in the past and are the building blocks of previously proposed two-cluster microarchitectures. Similarly configured 16-wide CTCP's have been studied [7, 20], but not with respect to the performance of dynamic cluster assignment options.

An example of the instruction and data routing for the baseline CTCP is shown in Figure 1. Notice that the cluster assignment for a particular instruction is dependent on its placement in the instruction buffer. The details of a single cluster are explored later in Figure 3.

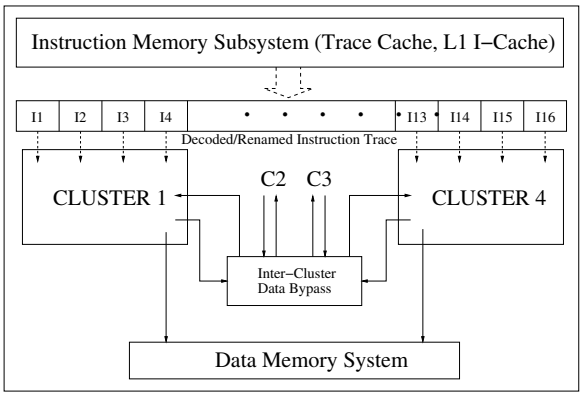

Figure 1. Overview of a Clustered Trace Cache Processor

C2 and C3 are clusters identical to Cluster 1 and Cluster 4.

2.1. Shared Components

The front-end of the processor (i.e. fetch and decode) is shared by all of the cluster resources. Instructions fetched from the trace cache (or from the instruction cache on a trace cache miss) are decoded and renamed before finally being distributed to their respective clusters. The memory subsystem components, including the store buffer, load queue, and data cache, are also shared.

Pipeline The pipeline for our baseline microarchitecture is shown in Figure 2. Three pipeline stages are assigned for instruction fetch (illustrated as one box). After the instructions are fetched, there are additional pipeline stages for decode, rename, issue, dispatch, and execute. Register file accesses are initiated during the rename stage. Memory instructions incur extra stages to access the TLB and data cache. Floating point instructions and complex instructions (not shown) also endure extra pipeline stages for execution.

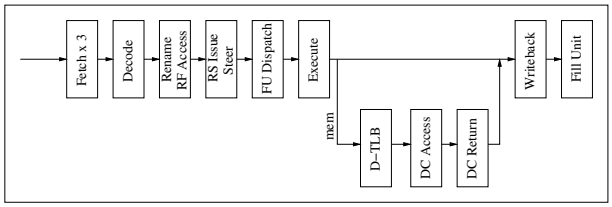

Figure 2. The Pipeline of the Baseline CTCP

Trace Cache The trace cache allows multiple basic blocks to be fetched with just one request [14, 15, 18]. The retiring instruction stream is fed to the fill unit which constructs the traces that consist of up to three basic blocks. When the traces are constructed, the intra-trace and intra-block dependencies are analyzed. This allows the fill unit to add bits to the trace cache line, which accelerates register renaming

and instruction steering [14].

2.2. Cluster Design

The execution resources modeled in this work are heavily partitioned. As shown in Figure 3, each cluster consists of five reservation stations which feed a total of eight special-purpose functional units. The reservation stations hold eight instructions and permit out-of-order instruction selection. The economical size reduces the complexity of the wake-up and instruction select logic while maintaining a large overall instruction window size [12].

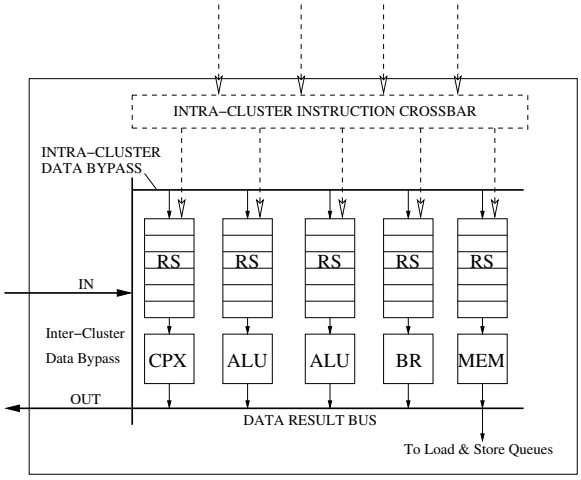

Figure 3. Design Details of One Cluster
There are eight special-purpose functional units per cluster: two simple integer units (*ALU*), one integer memory unit (*MEM*), one branch unit (*BR*), one complex integer unit (*CPX*), one basic floating point (FP), one complex FP, one FP memory (floating point units are not shown). There are five 8-entry reservation stations: one for the memory operations (integer and FP), one for branches, one for complex arithmetic, two for the simple operations.

Intra-cluster communication (i.e. forwarding results from the execution units to the reservation stations within the same cluster) is done in the same cycle as instruction dispatch. However, to forward data to a neighboring cluster takes two cycles. Forwarding beyond an adjacent cluster takes an additional two cycles for each cluster hop. This latency includes all of the communication and routing overhead associated with sharing inter-cluster data [13, 20]. The end clusters (clusters 1 and 4) do not communicate directly. There are no data bandwidth limitations between clusters in our work. Parcerisa et al. show that a point-to-point interconnect network can be built efficiently and is preferable to bus-based interconnects [13].

2.3. Cluster Assignment

The challenge to high performance in clustered microarchitectures is assigning instructions to the proper cluster. This includes identifying the proper destination cluster and then routing the instructions accordingly. With 16 instructions to analyze and four clusters from which to choose, picking the best execution resource is not straightforward.

Accurate dependency analysis is a serial process and is difficult to accomplish in a timely fashion. For example, approximately half of all result-producing instructions have data consumed by an instruction in the same cache line [1]. Some of this information can be preprocessed by the fill unit, but issue-time processing is also required. Properly analyzing the relationships is critical but costly in terms of pipeline stages. Any extra pipeline stages at issue time hurt performance when the pipeline refills after branch mispredictions and instruction cache misses.

Totally flexible routing is also a high-latency process. Instead, our baseline architecture steers instructions to a cluster based on the instruction's physical placement in the instruction buffer. Instructions are sent in groups of four to their corresponding cluster where they are routed on a smaller crossbar to their proper reservation station. This style of partitioning results in less complexity and fewer critical pipeline stages, but is restrictive in terms of issue-time flexibility and steering capabilities.

A large crossbar would permit instruction movement from any position in the instruction buffer to any of the clusters. In addition to the latency and complexity drawbacks, this option mandates providing enough reservation station write ports to accommodate up to 16 new instructions per cycle. Therefore, we concentrate on simpler, low-latency instruction steering options.

Assignment Options For comparison purposes, we look at the following dynamic cluster assignment options:

- **Issue Time:** Instructions are distributed to the cluster where one or more of their data inputs are known to be generated. Inter-trace and intra-trace dependencies are visible. In each cycle, a maximum of four instructions can be assigned to each cluster. In addition to simplifying the hardware, this limit balances the cluster workloads. The issue-time cluster assignment option is examined with two different latencies for dependency analysis, instruction steering, and routing: 1) the ideal case of zero latency and 2) four cycles of latency.
- **Friendly Retire Time:** This is the only previously proposed fill unit cluster assignment policy of which we are aware. Friendly et al. propose a fill unit reordering and assignment scheme based on intra-trace dependency analysis [7]. Their scheme assumes a front-end scheduler restricted to simple slot-based issue, as in our base model. For each issue slot, each instruction is checked for an intra-trace input dependency for the respective cluster. Based on these data dependencies, instructions are physically reordered within the trace.

3. CTCP Characteristics

The following characterization serves to highlight the cluster assignment optimization opportunities. The data is collected for our base trace cache processor which has a maximum trace line size of 16 instructions (more architecture details in Table 7).

3.1. Instruction Stream Analysis

Table 1 presents run-time trace cache characteristics for our benchmarks. The first metric (*% TC Instr*) is the percentage of all retired instructions that were fetched from the trace cache. Benchmarks with a large percentage of trace cache instructions benefit more from fill unit optimizations since instructions from the instruction cache are not optimized for the CTCP. *Trace Size* is the average number of instructions per trace line. When the fill unit does the intra-trace dependency analysis for a trace, this is the available optimization scope.

Table 1. Trace Cache Characteristics

	% TC Instr	Trace Size
bzip2	99.31	10.46
eon	72.63	10.21
gzip	91.36	11.79
perlbmk	86.43	10.91
twolf	78.03	10.32
vpr	86.67	11.10
Avg	85.74	10.80

Figure 4 breaks down the sources of the most critical input. The input data that arrives last is defined to be the most critical. This figure presents the percentage of instructions that had their most critical input arrive from the register file, the producer for RS1, and the producer for RS2. On average, 44% of the instructions obtain their critical input data from the register file, 31% receive their final data input from the producer for register RS1, and 25% from the producer for RS2.

Table 2 further analyzes the critical dependencies seen in data forwarding. The first column indicates the percentage of all critical data dependencies that are satisfied by data forwarding. On average, about 83% of dependencies are critical. Of those, 28% are inter-trace dependencies.

3.2. Impact of Dependency Latencies

Figure 5 presents the performance impact of eliminating certain dependency-related latencies. The bars labeled *No Fwd Lat* represent our base model with no data forwarding latency. If all inter-cluster forwarding could take place in the same cycle, performance would improve by 41.8%.

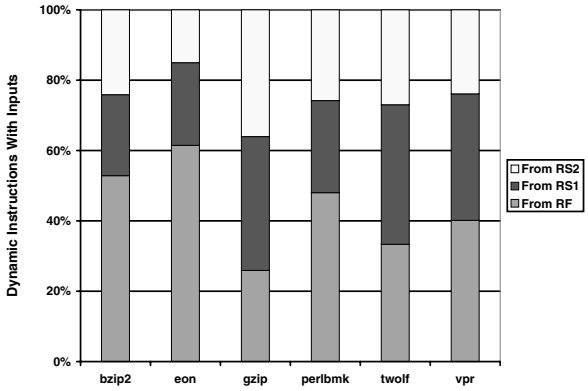

Figure 4. Source of Most Critical Input Dependency
From RS2: Critical input provided by the producer for input RS2. *From RS1:* Critical input provided by the producer for input RS1. *From RF:* Critical input provided by the register file.

Table 2. Critical Data Forwarding Dependencies

	% of all dep.'s that are critical	% of critical dep.'s that are inter-trace
bzip2	85.63%	29.69%
eon	86.58%	35.40%
gzip	80.94%	24.38%
perlbmk	86.11%	27.76%
twolf	78.58%	23.95%
vpr	82.32%	25.84%
Avg	83.36%	27.84%

Next, only the last forwarded value to an instruction is given a latency of zero cycles (*No Crit Fwd Lat*). This improves performance by 37.2%[1]. This result means that most of the improvement provided by eliminating all forwarding latencies is achieved just by eliminating the latency for the last arriving data input. This observation is exploited in the proposed cluster assignment scheme.

The speedup due to eliminating the register file read latency is presented in Figure 5 (*No RF Lat*) and has almost no effect on overall performance. In fact, register file latencies between zero and 10 cycles have no impact on performance. This is due to the abundance and critical nature of in-flight instruction data forwarding seen in a very wide-issue processor.

Finally, the speedups due to only removing inter-trace and intra-trace data forwarding latencies are also presented in Figure 5 (*No Inter-Trace Lat* and *No Intra-Trace Lat*). Reducing the inter-trace forwarding latency to zero cycles improves performance by 15.5% while a similar theoretial improvement for intra-trace forwarding results in a 17.7% performance gain.

Table 2 shows a large percentage of intra-trace depen-

[1]For *bzip2*, the branch predictor accuracy is sensitive to the rate at which instructions retire and the "better" case with no data forwarding latency actually leads to a 3.5% increase in branch mispredictions and worse performance.

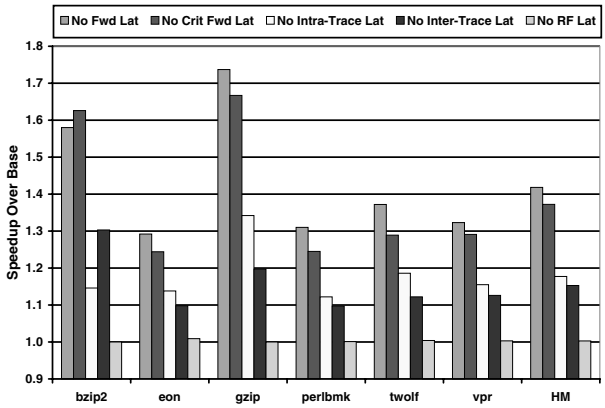

Figure 5. Expected Speedup Removing Certain Latencies
The base processor described in Section 5.1 is used, where the default inter-cluster forwarding latency is two cycles and the default intra-cluster forwarding latency is zero cycles. Also, note that the y-axis starts at 0.90. However, no speedups fall below 1.00.

dencies. Despite this, the speedup gained by removing the inter-trace forwarding latency is similar to that of removing intra-trace forwarding latency. In the case of *bzip2*, removing inter-trace data forwarding latencies is more beneficial than intra-trace.

These results imply that an inter-trace dependency is more critical to advancing execution than an intra-trace dependency. One explanation is that the producer of an inter-trace dependency is by definition in a separate trace and must issue in a later cycle, which is not the case with intra-trace dependencies. Also, all inter-trace dependencies are also inter-basic block dependencies. An input from another basic block often triggers a chain of computations within a basic block.

3.3. Resolving Inter-Trace Dependencies

The fill unit can accurately determine intra-trace data dependencies. Since a trace is an atomic unit in the trace cache, the same intra-trace instruction data dependencies will exist when the trace is later fetched. However, incorporating inter-trace dependencies at retire time is essentially a prediction of issue-time dependencies from unknown producers, which may exist thousands or millions of cycles in the future.

This problem presents an opportunity for an execution history-based mechanism to predict the source clusters or producers for instructions with inter-trace dependencies. Table 3 examines how often an instruction's forwarded data comes from the same producer instruction. For each static instruction, the program counter of the last producer is tracked for each source register (RS1 and RS2). The second and third columns of the table show that an instruction's data forwarding producer is the same for RS1 97.1% of the time and the same for RS2 94.5% of the time.

Table 3. Frequency of Repeated Forwarding Producers

	All		Critical Inter-trace	
	Input RS1	Input RS2	Input RS1	Input RS2
bzip2	97.44%	97.66%	89.30%	91.17%
eon	93.83%	89.84%	85.79%	73.34%
gzip	98.14%	99.02%	92.93%	96.04%
perlbmk	97.78%	93.79%	90.83%	79.27%
twolf	96.69%	90.78%	87.09%	76.40%
vpr	98.53%	96.06%	95.64%	91.67%
Average	97.07%	94.52%	90.26%	84.65%

The last two columns isolate the critical producers of only inter-trace consumers. These percentages are expectedly lower, but producers are still repeated for 90.3% of the critical inter-trace RS1 inputs and for 84.7% of the critical inter-trace RS2 inputs. Therefore, a basic prediction mechanism could work with a high degree of success.

4. Retire-Time Cluster Assignment

This section presents the background, motivation, and details for our feedback-directed, retire-time (FDRT) cluster assignment scheme. The proposed cluster assignment enhancements take advantage of the dynamic optimization opportunities of a long-latency fill unit. The FDRT strategy improves retire-time cluster assignment by assigning instructions to a cluster based on both intra-trace *and* inter-trace dependencies. In addition, the critical instruction input is considered.

The fill unit is available at instruction retirement to analyze the instruction dependencies. Instructions are physically reordered to match the desired cluster assignments. Since there are no issue-time decisions or routing, the instructions must be physically placed within the trace so that they issue directly to the desired execution cluster when the trace is later fetched from the trace cache.

The physical reordering does not affect the logical ordering of the instructions. Instructions retire in the same order regardless of their physical position in the trace cache line. The logical order is marked in the trace cache line by the fill unit. Though this adds some complexity, techniques such as precomputing register renaming information and predecoding help to mitigate complexity issues [7]. The important result is that physical reordering reduces inter-cluster communications while maintaining low-latency, complexity-effective issue logic.

Previously, a fill unit latency of up to 10 cycles was shown to have negligible effects on overall performance [14]. In our environment, simulations have shown that a latency of 1000 cycles does not significantly impact FDRT performance [2].

4.1. Chaining Instructions

One goal of FDRT assignment is to speculatively assign dependent inter-trace instructions to the same cluster. This not only requires identifying the dependent pairs but also identifying the cluster to which they should be assigned. To accomplish this, designated producers of inter-trace dependencies are provided with a suggested destination cluster. Later on, the fill unit will attempt to accommodate this suggestion. These instructions pass this cluster value to their inter-trace consumers and those consumers pass it to their inter-trace consumers and so on. In this manner, key producers and their subsequent inter-trace consumers can be routed to the same cluster.

This chaining forces intra-cluster data forwarding between inter-trace dependencies, and in the process a *cluster chain* of instructions with inter-trace dependencies is created. The first instruction of the cluster chain is called a *leader*. The subsequent links of the chain are referred to as *followers*.

Physically reordering instructions at retire time based on inter-trace data dependency history can cause more inter-cluster data forwarding than it eliminates. The same trace of instructions can be reordered in a different manner each time the trace is constructed. Producers may shift from one cluster to another, never allowing consumers to accurately gauge the cluster from which their input data will be produced.

To eliminate this effect, when an instruction is assigned to a cluster chain as a leader or a follower, its suggested execution cluster never changes. The idea is to permanently *pin* a leader to one cluster and not permit the leader or its followers to change their chain cluster. The criteria for selecting inter-trace dependency cluster chain leaders and followers are presented in Table 4.

Table 4. Leader and Follower Criteria

Conditions to become a **cluster chain leader**:
1. Cannot already be a leader or a follower
2. Forward data to inter-trace consumer(s)
Conditions to become a **cluster chain follower**:
1. Not already a leader or follower
2. Producer is a leader or follower
3. Producer is from different trace
4. Producer provides last input data

The key aspect to these guidelines is that only inter-trace dependencies are considered. Placing instructions with intra-trace dependencies on the same cluster is easier and accurate. Therefore, instructions with intra-trace dependences do not require cluster chaining support to establish dependencies.

4.2. Dynamic Instruction Feedback

The trace cache framework provides a unique instruction feedback loop. Instructions are fetched from the trace cache, executed, and then compiled into new traces. This instruction feedback enables run-time, instruction-level profiling. By associating a few bits with each instruction in the trace cache[2], a dynamic execution history is built for each instruction. This execution data remains associated with the instruction as it travels through the pipeline. This method of run-time execution profiling is very effective as long as the trace lines are not frequently evicted from the trace cache. Zyuban et al. suggest placing static cluster assignments in the trace cache, but do not provide details, results, or analysis [20].

In our enhanced clustered trace cache processor microarchitecture, run-time per-instruction information is used to provide inter-trace dependency information to the fill unit. There are two fields added to the trace cache storage:

- **Chain Cluster:** This field holds the chain cluster number. Producers forward this two-bit cluster number along with its result to consumers.
- **Leader/Follower Value:** This two-bit field indicates whether the instruction is a leader, a follower, or neither.

4.3. Cluster Assignment Strategy

The fill unit must weigh intra-trace information along with the inter-trace feedback from the trace cache instruction histories. Table 5 summarizes our proposed cluster assignment policies. The inputs to the strategy are: 1) the presence of a critical intra-trace dependency, 2) the cluster chain status, and 3) the presence of intra-trace consumers. The fill unit starts with the oldest instruction and progresses in logical order to the youngest instruction.

Shown as Option *A* in Table 5, the fill unit attempts to place instructions that have only an intra-trace dependency on the same cluster as its producer. If four instructions have already been assigned to this cluster, there is an attempt to assign the instruction to the neighbor of the chain cluster. For an instruction with just an inter-trace chain dependency (Option *B*), the fill unit attempts to place the instruction on the chain cluster (which is found in the Chain Cluster trace profile field) or a cluster that neighbors the chain cluster.

An instruction can have both an intra-trace dependency and a chain inter-trace dependency (Option *C*) if the critical input changes or an instruction is built into a new trace. When an instruction has both a chain cluster and an intra-trace producer, the chain cluster takes precedence (although

[2]Cacti 2.0 [17] shows that an additional byte per instruction in a trace cache line does not change the fetch latency of the trace cache.

Table 5. FDRT Cluster Assignment Strategy

Dependency type	Option A	Option B	Option C	Option D	Option E
	if...	if...	if...	if...	if...
1. Has intra-trace producer:	yes	no	yes	no	no
2. Is inter-trace chain member:	no	yes	yes	no	no
3. Has intra-trace consumer:	-	-	-	yes	no
	then...	then...	then...	then...	then...
Resulting Cluster Assignment Priority:	1. producer 2. neighbor 3. skip	1. chain 2. neighbor 3. skip	1. chain 2. producer 3. neighbor 4. skip	1. middle 2. skip	1. skip

our simulations show that it does not matter which gets precedence). If there are no instruction slots available for this cluster, the intra-trace producer's cluster is the next target. Finally, neighbors of the chain cluster are tried.

If an instruction has no dynamically forwarded input data but does have an intra-trace output dependency (Option *D*), it is assigned to a middle cluster, reducing potential forwarding distances.

Instructions are skipped if they have no critical or detectable producers or consumers (Option *E*), or if they cannot be assigned to a cluster near their producer (lowest priority assignment for Options *A*-*D*). These instructions are later assigned to the remaining slots using Friendly's method.

5. Performance Evaluation

5.1. Simulation Methodology

The clustered trace cache processor is evaluated using integer benchmarks from the SPEC CPU2000 suite [19]. The benchmarks and their respective inputs are presented in Table 6. The MinneSPEC reduced input set [10] is used when available. Otherwise, the SPEC *test* input is used.

Table 6. SPEC CINT2000 Benchmarks

Benchmark	Input Src	Inputs
bzip2	MinneSPEC	lgred.source 1
eon	SPEC test	chair.control.kajiya chair.camera chair.surfaces ppm
gzip	MinneSPEC	smred.log 1
perlbmk	MinneSPEC	mdred.makerand.pl
twolf	MinneSPEC	mdred
vpr	MinneSPEC	mdred.net small.arch.in -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2

The benchmark executables are precompiled Alpha binaries available with the SimpleScalar 3.0 simulator [3]. The target architecture for the compiler is a four-way Alpha 21264, which in many ways is convenient for a clustered architecture with four-wide clusters. All benchmarks are run for 100 million instructions.

Six SPEC CPU2000 benchmarks are selected for in-depth analysis and presentation. They are chosen based on their sensitivity to data forwarding latency to underscore the potential of CTCP cluster assignment optimizations. The benchmarks that show the most performance improvement potential when using a zero-cycle inter-cluster latency are chosen. Performance comparison for the entire integer suite along with 14 MediaBench programs are presented in Section 5.6.

To perform the simulations, a detailed, cycle-accurate microprocessor simulator is interfaced to the functional simulator from the SimpleScalar 3.0 simulator suite (*sim-fast*) [3]. The architectural parameters for the simulated base microarchitecture are shown in Table 7.

Table 7. Baseline Architecture Configuration

Data memory	
L1 Data Cache:	4-way, 32KB, 2-cycle access
L2 Unified cache:	4-way, 1MB, +8 cycles
Non-blocking:	16 MSHRs and 4 ports
D-TLB:	128-entry, 4-way, 1-cyc hit, 30-cyc miss
Store buffer:	32-entry w/load forwarding
Load queue:	32-entry, no speculative disambiguation
Main Memory:	Infinite, +65 cycles

Fetch Engine	
Trace cache:	2-way, 1K-entry, 3-cycle access
L1 Instr cache:	4-way, 4KB, 2-cycle access
Branch Predictor:	16k-entry gshare/bimodal hybrid
BTB:	512 entries, 4-way

Execution Cluster

· Functional unit	#	Exec. lat.	Issue lat.
Simple Integer	2	1 cycle	1 cycle
Simple FP	2	3	1
Memory	1	1	1
Int. Mul/Div	1	3/20	1/19
FP Mul/Div/Sqrt	1	3/12/24	1/12/24
Int Branch	1	1	1
FP Branch	1	1	1

· Inter-Cluster Forwarding Latency: 2 cycles per hop
· Register File Latency: 2 cycles
· 5 Reservation stations
· 8 entries per reservation station
· 2 write ports per reservation station

· 128-entry ROB · Fetch width: 16
· Decode width: 16 · Issue width: 16
· Execute width: 16 · Retire width: 16

5.2. Performance Analysis

Figure 6 presents the execution time speedups normalized to our base architecture for different dynamic cluster assignment strategies. Friendly's method improves performance by 3.1%[3]. The proposed feedback-directed, retire-time (FDRT) cluster assignment strategy provides an 11.5% improvement. This improvement in performance is due to enhancements in both the intra-trace and inter-trace aspects of cluster assignment.

The two retire-time instruction reordering strategies are also compared to issue-time instruction steering in Figure 6. In one case, instruction steering and routing is modeled with no latency (labeled as *No-lat Issue-time*) and in the other case, four cycles are modeled (*Issue-time*). The results show that latency-free issue-time steering is the best overall option studied, with a 17.2% improvement over the base. This is the best performance method for all benchmarks except *bzip2*, where the FDRT strategy for cluster assignment is preferable. When applying a four-cycle latency, issue-time steering is only preferable for half of the benchmarks, and the average performance improvement is comparable to FDRT cluster assignment. Section 5.6 illustrates a more distinct advantage for FDRT assignment over issue-time for a wider array of benchmarks.

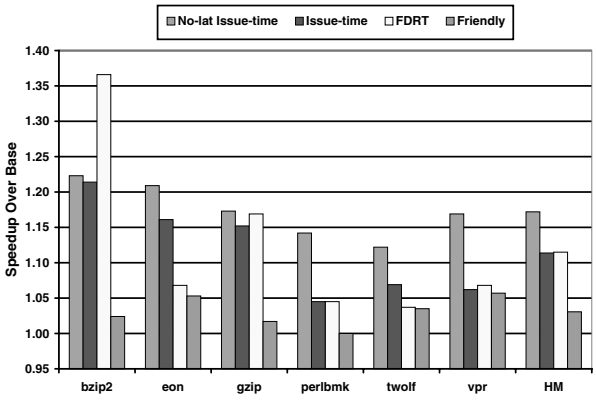

Figure 6. Speedup Due to Cluster Assignment Strategy

Note that the y-axis starts at 0.95. However, no bar is below 1.00.

The core reasons that FDRT assignment provides a performance boost over the Friendly and baseline assignment methods are an increase in intra-cluster forwarding and a reduction in average data forwarding distance. Table 8a presents the changes in intra-cluster forwarding. On average, both CTCP retire-time cluster assignment schemes increase the amount of same-cluster forwarding to above 50%, with FDRT assignment doing better on average.

The inter-cluster distance is the aspect of cluster assignment that has the biggest impact on performance (Table 8b).

[3]All speedup averages are computed using the harmonic mean.

For every benchmark, the retire-time instruction reordering schemes are able to improve upon the average forwarding distance. In addition, the FDRT scheme always provides shorter overall data forwarding distances than the Friendly method. This is a result of funneling producers with no input dependencies to the middle clusters and placing consumers as close as possible to their producers.

Table 8. Data Forwarding For Critical Inputs

a. Percentage of Intra-Cluster Forwarding

	Base	Friendly	FDRT
bzip2	39.79%	60.84%	79.54%
eon	33.73%	52.83%	51.35%
gzip	32.94%	53.91%	58.25%
perlbmk	44.95%	58.36%	62.01%
twolf	47.83%	56.91%	58.92%
vpr	38.67%	58.70%	59.58%
Average	39.65%	56.93%	61.61%

b. Average Data Forwarding Distance

	Base	Friendly	FDRT
bzip2	0.83	0.58	0.24
eon	0.96	0.73	0.70
gzip	0.94	0.77	0.56
perlbmk	0.78	0.62	0.49
twolf	0.73	0.66	0.56
vpr	0.92	0.70	0.57
Average	0.86	0.67	0.52

The *critical input* is the data input that arrives last. If there is only one input for the instruction, then it is the critical input. *Distance* is the number of clusters traversed by forwarded data.

For the program *eon*, the Friendly strategy provides a higher intra-cluster forwarding percentage than FDRT without resulting in higher performance. The primary reason is that FDRT reduces the average data forwarding distance even with the presence of extra inter-cluster forwarding.

5.3. Improvements Over Prior Method

The FDRT method of instruction reordering and cluster assignment has several advantages over Friendly's previously proposed CTCP instruction reordering strategy. The most obvious improvement is the inclusion of inter-trace information gathered in the trace cache instruction profiles. The importance of accounting for these dependencies is illustrated in Figure 5.

Additionally, the variable data forwarding latencies between clusters are taken into account by the FDRT instruction reordering. The inter-cluster forwarding latency is variable based on the distance between the communicating clusters. Therefore, the FDRT strategy funnels instructions to the middle clusters. This reduces the amount of forwarding that must span two and three clusters. If the Friendly instruction placement is modified such that it assigns a majority of the instructions to the middle clusters, the average performance improvement increases to 4.7% [2].

Finally, Friendly's strategy examines each instruction slot and looks for a suitable instruction while the FDRT

method looks at each instruction and finds a suitable slot. This subtle difference accounts for some performance improvement as well. Additional analysis [2] shows that isolating the intra-trace heuristics from the FDRT strategy results in a 5.7% improvement by itself, compared to the 3.1% for Friendly's method. The remaining performance improvement generated by FDRT assignment comes from incorporating inter-trace dependency information.

5.4. FDRT Assignment Analysis

Figure 7 is a breakdown of instructions based on their FDRT assignment strategy option from Table 5. On average, 37% of instructions have only a critical intra-trace dependency, while 18% of the instructions have just an inter-trace chain dependency. Only 9% of the instructions have both a chain inter-trace dependency and a critical intra-trace dependency. These three categories of instructions (64% of all instructions) all have identifiable producers and are placed on a cluster near their producers.

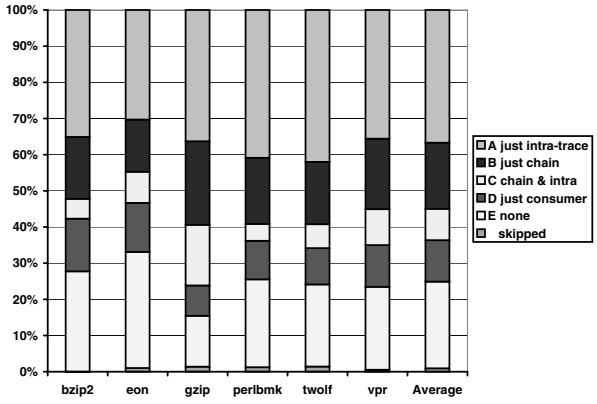

Figure 7. FDRT Critical Input Distribution

The letters A-E correspond to the lettered options in Table 5.

Approximately 11% of the instructions had no input dependencies but did have an intra-trace consumer. These producer instructions are assigned to a middle cluster where their consumers will be placed on the same cluster later. Only a very small percentage (less than 1%) of instructions with identified input dependencies are initially skipped because there is no neighbor cluster for assignment. Finally, a large percentage of instructions (around 24%) are determined to not have a critical intra-trace dependency, chain inter-trace dependency, or intra-trace consumer. Most of these instructions do have data dependencies, but they did not require data forwarding or did not meet the chain criteria.

5.5. Pinning Leaders to a Cluster

This section examines the effects of permanently pinning chain leaders to a cluster. Recall that without this pinning, both chain leaders and followers could become part of any chain including their own. This could result in constantly changing chain cluster assignments where chain members are essentially chasing a moving target. We call this effect *cluster migration*.

Instruction cluster migration is quantified in Table 9 for the presented version of FDRT (*Pinning*) and for a version that does not pin chains to a cluster (*No Pinning*). On average, without pinning the chain to a cluster, 5.80% of all dynamically encountered instructions are physically reordered such that their assigned cluster is different from their previous dynamic invocation. The largest percentage is 8.9% for *twolf*. However, once the leaders are pinned, the overall percentage of instructions experiencing cluster migration is reduced to an average of 4.25%. More importantly, the percentage of chain instructions that migrate is reduced by 41%.

Table 9. Instruction Cluster Migration

	All Instr. Pinning	All Instr. No Pinning	All Instr. Reduction	Chain Instr. Reduction
bzip2	0.35%	0.98%	63.86%	62.73%
eon	5.94%	8.27%	28.20%	52.97%
gzip	5.97%	8.26%	27.68%	30.65%
perlbmk	3.77%	3.59%	-4.98%	19.30%
twolf	5.08%	8.92%	42.99%	56.29%
vpr	4.36%	4.77%	8.48%	23.94%
Average	4.25%	5.80%	27.71%	40.98%

Another interesting observation from this table is the increase in cluster migration for *perlbmk* while pinning the chains. In this case, the migration is reduced for chain instructions, which is the intended effect. However, for this benchmark, the scheduling of the non-chain instructions proves to be less stable when pinning the chain leaders.

The purpose of pinning the instructions is to reduce instructions from oscillating between clusters and to increase the amount of intra-cluster forwarding for critical data dependencies. The change in intra-cluster forwarding can be seen in Table 10. Overall, pinning instructions increases the amount of same-cluster critical data forwarding for four out of six benchmarks and by a small percentage overall (60.51% to 58.57%). Only *bzip2* sees a significant change in intra-cluster data forwarding with pinning.

5.6. Strategy Robustness

Figure 8 presents the speedup for our main cluster assignment strategies with different cluster properties. The first modification is to use a mesh network, where clus-

Table 10. Intra-Cluster Critical Data Forwarding During Cluster Migration

	With Pinning	No Pinning
bzip2	77.48%	66.69%
eon	49.72%	50.88%
gzip	56.03%	55.03%
perlbmk	65.32%	65.36%
twolf	57.51%	57.13%
vpr	57.01%	56.34%
Average	60.51%	58.57%

ters 1 and 4 communicate directly. This network eliminates three-cluster communication and is advocated by Parcerisa et al. [13]. The second modification is to eliminate one cycle of data forwarding latency. The third modification is to cut the execution resources in half (instruction cache, data cache, branch predictor, and TLB remain the same size). The issue-time latency is reduced to two cycles since there are now only eight instructions to analyze, steer, and route.

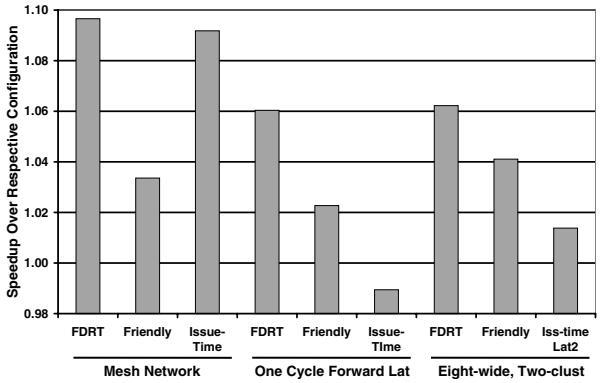

Figure 8. Speedups For Other Cluster Configurations
The speedups for each group is relative to their respective base model. Note that the y-axis begins at 0.98 and not 1.00.

The presented cluster architecture variations lead to a reduction in the absolute speedup numbers for all strategies compared to the original architecture. However, the proposed strategy still provides superior performance over the cluster assignment alternatives without any type of architecture-specific fine-tuning of FDRT assignment. The relative strength of the FDRT strategy over issue-time assignment increases in all cases compared to performance seen for the base architecture. However, the advantage over the Friendly method is not as significant.

The issue-time performance in the mesh network does relatively well because the average forwarding distance is reduced and the processor setup is now more similar to the assignment strategy's intended architecture. On the other hand, a smaller forwarding latency (with no mesh) does not allow the issue-time strategy to overcome its latency drawbacks, resulting in an overall performance drop.

Figure 9 presents the average speedup for all 12 SPEC CPU2000 integer benchmarks as well as for 14 MediaBench benchmarks [11] that are used in previous four-cluster work [13]. For both benchmark suites, FDRT cluster assignment provides over twice as much performance improvement as the Friendly method. In addition, retire-time assignment is still shown to be preferable to realistic issue-time instruction steering.

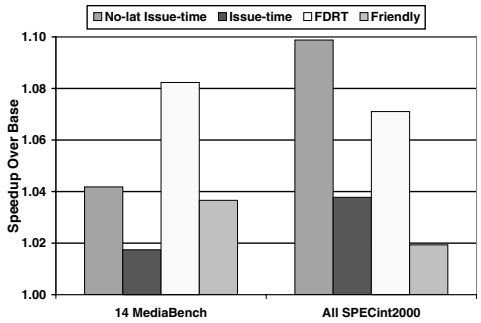

Figure 9. Dynamic Cluster Assignment Speedups for SPECint2000 and MediaBench Benchmark Suites
All benchmarks are run for 100M instructions or until completion. Default inputs are used for MediaBench. MinneSPEC and test inputs are used for SPECint2000.

Not all SPEC programs are presented throughout this paper because some see only modest improvement regardless of the dependency-based scheduling scheme. These programs are often limited by some other aspect of program execution, such as memory or branch prediction. The issue-time assignment presents more problems for programs with a high percentage of branch mispredictions and instruction cache misses because of the extra latency at issue time. However, FDRT assignment does not slow down any of these programs. The average speedup considering all SPEC CPU2000 integer benchmarks is still a healthy 7.1% while issue-time improvement is 3.8%.

The most interesting result for the MediaBench programs is that FDRT scheduling (8.2% improvement) performs better than latency-free issue-time scheduling (4.2%). In this case, the latency-free issue-time heuristic results in better performance for nine progams, but is worse than FDRT for the other five programs. This includes slowing down three of the programs compared to the base, which does not happen with FDRT scheduling. Additional characterization and analysis for MediaBench and the complete SPEC integer suite are available [2].

6. Summary

In this work, we present a retire-time cluster assignment scheme for clustered trace cache processors. Utilizing a trace cache, instruction-level feedback is used to capture inter-trace dependency information. The fill unit combines

this previously unavailable information with intra-trace dependency information to assign instructions to clusters.

For six selected SPEC CPU2000 integer benchmarks, the proposed feedback-directed, retire-time (FDRT) cluster assignment strategy reduces the inter-cluster data forwarding from 60% to 38% while also reducing the average cluster forwarding distance by 40%. When incorporating inter-trace dependency feedback into the retire-time cluster assignment process, an 11.5% performance improvement over the base architecture is possible. The performance improvement is significantly higher than the 3.1% improvement seen using a previously proposed CTCP cluster assignment scheme (up to 4.7% improvement with a minor adjustment).

For a wider range of programs (all SPEC CPU2000 integer and MediaBench), FDRT cluster assignment leads to a 7.1% and 8.2% performance improvement, respectively. This is significantly higher than issue-time cluster assignment (3.8% and 1.7%) or Friendly's retire-time strategy (1.9% and 3.7%).

FDRT cluster assignment is also studied for other cluster configurations. Without modifications, the proposed strategy maintains a clear advantage over issue-time options as well as the previous CTCP retire-time option for cluster architectures with mesh interconnects, one-cycle forward latencies, and two four-wide clusters. This demonstrates the robustness and potential usefulness of feedback-directed, retire-time cluster assignment.

7. Acknowledgments

The authors would like to thank the anonymous reviewers for their suggestions and insight which greatly improved this paper. This research is partially supported by the National Science Foundation under grant number 0113105, and by AMD, Intel, IBM, Tivoli and Microsoft Corporations.

References

[1] R. Bhargava and L. K. John. Latency and energy aware value prediction for high-frequency processors. In *16th International Conference on Supercomputing*, pages 45–56, June 2002.

[2] R. Bhargava and L. K. John. Cluster assignment strategies for a clustered trace cache processor. Technical Report TR-030331-01, The University of Texas at Austin, Laboratory For Computer Architecture, Mar 2003.

[3] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical report, University of Wisconsin, Madison, WI, 1997.

[4] R. Canal, J.-M. Pacerisa, and A. Gonzalez. A cost-effective clustered architecture. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 160–168, Oct 1999.

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

[6] M. Franklin. *The Multiscalar Architecture*. PhD thesis, Univ. of Wisconsin-Madison, 1993.

[7] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache processors. In *31st International Symposium on Microarchitecture*, pages 173–181, Dallas, TX, November 1998.

[8] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10(14), Oct 1996.

[9] Q. Jacobson and J. E. Smith. Instruction pre-processing in trace processors. In *International Symposium of High Performance Computer Architecture*, Jan 1999.

[10] A. KleinOsowski and D. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.

[11] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. *IEEE Micro*, vol. 30, no. 1, pp. 330-335, Dec. 1997.

[12] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *24th International Symposium on Computer Architecture*, pages 206–218, June 1997.

[13] J.-M. Parcerisa, J. Sahuquilla, A. Gonzalez, and J. Duato. Efficient interconnects for clustered microarchitectures. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 291–300, Sep. 2002.

[14] S. J. Patel, D. H. Friendly, and Y. N. Patt. Critical issues regarding the trace catch fetch mechanism. Technical report, University of Michigan, 1997.

[15] A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. U.S. Patent Number 5,381,533, 1994.

[16] A. G. R. Canal, J-M. Parcerisa. Dynamic cluster assignment mechanisms. In *6th International Symposium on High Performance Computer Architecture*, pages 132–142, Jan 2000.

[17] G. Reinman and N. Jouppi. An integrated cache timing and power model, 1999. COMPAQ Western Research Lab.

[18] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *29th International Symposium on Microarchitecture*, pages 24–34, Dec. 1996.

[19] Standard Performance Evaluation Corporation. SPEC CPU2000 Benchmarks. http://www.spec.org/osg/cpu2000/.

[20] V. V. Zyuban and P. M. Kogge. Inherently lower-power high-performance superscalar architectures. *IEEE Transactions on Computers*, 50(3):268–285, Mar 2001.