Dynamically Matching ILP Characteristics Via a Heterogeneous Clustered Microarchitecture

Lei Chen, David H. Albonesi, Steven Dropsho†

Department of Electrical and Computer Engineering
†Department of Computer Science
University of Rochester
Rochester, NY 14627

Abstract

Applications vary in the degree of instruction level parallelism (ILP) available to be exploited by a superscalar processor. The ILP can also vary significantly within an application. On one end of the microarchitecture space are monolithic superscalar designs that exploit parallelism within an application. At another end of the spectrum are clustered architectures having many simple cores that can be clocked at a higher frequency than a comparable monolithic design. A disadvantage of the clustered design is the cost to transmit results between clusters which potentially limits the performance even in high ILP applications if instructions are not mapped carefully to minimize cross communication.

In this paper, we propose an approach that incorporates the strengths of the prior two by clustering multiple integer ALUs in an asymmetric fashion. In one cluster, a 6-way out-of-order pipeline efficiently executes code having high ILP. In another cluster, a simpler, but deeper, 2-way pipeline running at twice the frequency speeds up regions of code having little ILP. We use the parallelism metrics gathered from the dynamic Data Dependence Tracking (DDT) mechanism to dynamically steer instruction windows to a cluster. We demonstrate that the heterogeneous cluster design can improve performance by up to 30% over a monolithic 8-way superscalar processor.

1 Introduction

A fundamental tradeoff in processor microarchitecture is in achieving the best balance between maximizing instructions per cycle (IPC) performance and clock frequency. Due to the diverse nature of application programs, in particular with regards to their instruction-level parallelism (ILP) and branch and memory behavior, the effectiveness of complex hardware for exploiting ILP varies considerably. Recent *adaptive* approaches attempt to exploit this variability by *dynamically* adjusting the complexity of major hardware structures (issue queue, caches, *etc.*) to match these varying demands. In this paper, we investigate an alternative approach, that of providing two *static* pipeline designs that are optimized for different ILP characteristics and steering *windows* of instructions to one pipeline or the other.

Although prior clustered designs have been proposed and even implemented (*e.g.*, in the Alpha 21264 microprocessor [8]), these are *homogeneous* in nature and are used primarily to increase frequency at the expense of reducing IPC. The latter stems from the inter-cluster communication delays that are incurred when register values are passed among clusters. Our *heterogeneous* clustering approach is rather to tailor each cluster to particular application ILP characteristics, and to use only one cluster at a time. By switching between clusters at a coarse grain, and sending values to the disabled cluster's register file throughout execution to keep it current, we largely avoid the inter-cluster communication penalties of the homogeneous cluster design, and are able to tailor the pipeline to the application.

The rest of this paper is organized as follows. In the next section, we describe the heterogeneous cluster design. We provide details of the pipeline design by which the frequency is increased at the expense of additional pipeline stages. Our methodology is described in Section 3, and our results in Section 4. We discuss related work in Section 5, and conclude in Section 6.

2 Heterogeneous Cluster Microarchitecture

To improve performance, general-purpose processor microarchitectures dedicate considerable resources in order to exploit ILP. The added circuit complexity can potentially impact the cycle time of the processor, or increase latency in datapaths, thereby mitigating some of the obtained IPC improvement. A balanced design makes trade-offs between the increased complexity and cycle time/latency to maximize overall performance.

Unfortunately, application phases with low ILP must pay a higher delay cost than what may be achieved with a simpler execution core matching the lower ILP. The microarchitecture we describe here has two execution cores, one *wide* and one *narrow*. The wide core runs at a base frequency and improves performance through exploiting parallelism. The narrow core runs at twice the frequency of the wide core (but is more deeply pipelined) and improves performance by executing code having little ILP faster (due to its higher throughput on narrow sustained streams of in-

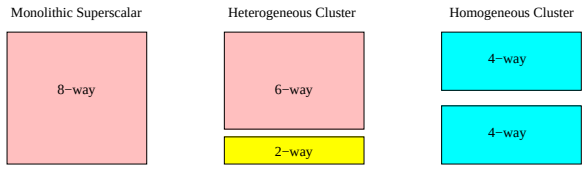


Figure 1. Architecture organizations

structions) than could be achieved in the wide core.

The two types of cores can be considered clusters and share some similarity with clustered microarchitectures. The design philosophy in a basic clustered organization is to develop simple cores that run at a high frequency and instantiate multiple copies on the chip for parallelism. We refer to traditional cluster design as a *homogeneous clustered* architecture because all the cores are identical. A performance bottleneck in such an architecture is often the cost of transmitting results between the cores. In the design we propose here, the cores can be considered "specialists" with one supporting high ILP computations and the other supporting low ILP computations. We refer to this type of design as a *heterogeneous clustered* architecture.

Figure 1 compares the three basic architectures: monolithic superscalar, our heterogeneous cluster design, and a traditional homogeneous cluster design. In the figure, larger blocks indicate a more complex design and slower clock frequency (though relative differences are not proportional to block size). Inter-block crossings imply a delay penalty. Intuitively, while the homogeneous cluster may permit the cores to run at the highest possible clock rate, phases with high degrees of parallelism may suffer a performance penalty due to frequent inter-cluster communication. Conversely, the monolithic superscalar design has nominal inter-ALU communication overhead to support high degrees of parallelism efficiently, but due to its lower clock frequency it is less efficient on instruction streams having little parallelism.

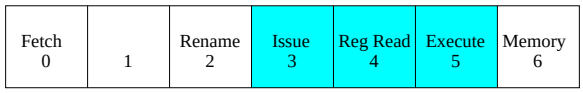
The proposed heterogeneous cluster architecture attempts to strike a balance between the two extremes by having cores target particular types of computation. By matching windows of instructions to the best core type, a heterogeneous cluster architecture can improve overall performance through dynamically exploiting either parallelism or high sustained throughput on narrow instruction streams.

2.1 Pipeline stages in the heterogeneous clusters

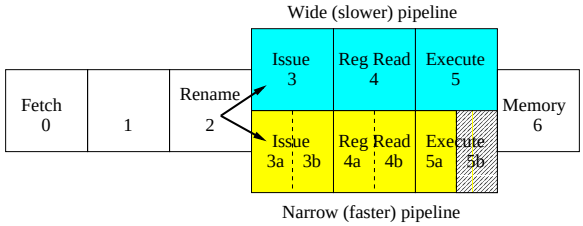
The narrower pipeline is simpler than the wider pipeline and can run at a higher clock rate. Shown in Figure 2(a) is a basic pipeline based on the SimpleScalar toolset [2]. Shown in Figure 2(b) is the heterogeneous cluster organization. The clusters share the first three stages and the final stage. The middle stages *Issue*, *Register Read*, and *Execute* are the same in the wide (6-way) pipeline as the base pipeline. However, in the narrow (2-way) pipeline each of these three stages are further pipelined into an additional stage, but running at twice the frequency. Steering logic streams a window of instructions to either the wide pipeline or the narrow pipeline.

Each pipeline has a copy of the register file. Results generated in one pipeline are copied to the other register file after a delay to transport the data to the other pipeline.

To double the clock rate, the stages can be partitioned in the narrow pipeline in the following manner. The issue logic can be partitioned into a wake up stage and a select stage as described in [7]. The register file access is pipelined into two stages. In the execute stage, the ALUs are designed to run at twice the base frequency. This would require fast ALU circuit structures such as those used in the Pentium 4 microprocessor. Since simple operations such as addition and logical operations require only a single cycle, the stage *5b* performs no useful work other than to resync the instruction back to the shared pipeline clock. However, no synchronization is required due to the fact that both clocks are derived from the same source, and one is an integral multiple of the other. Note that we present here one possible example of a heterogeneous design. We have made the simplifying assumption that doubling the clock of these pipeline stages requires doubling the number of stages, even with the pipeline width reduced by two-thirds. In actuality, fewer, or perhaps more, stages may be required. With our assumption of twice the number of stages at twice the frequency, the branch mispredict penalty is the same (in absolute time, not "cycles") in both pipelines.



(a) Basic architecture pipeline



(b) Heterogeneous cluster pipeline

Figure 2. Pipeline structures

2.2 Instruction steering

In this study, we use two mechanisms to steer instructions to one cluster or the other. In typical cluster architectures, all clusters may be used simultaneously to increase parallelism so steering decisions are made on an instruction by instruction basis. In our heterogeneous design, we select a cluster to best match the parallelism across a *window* of instructions. Thus, steering decisions are made for groups

of instructions. As this group of instruction executes on one cluster, results are written to its register file as well as to that of the other cluster (but delayed). Thus, delays due to inter-cluster register communication can occur only on window boundaries. Even a multi-cycle delay in updating results between the pipelines has a tolerable performance degradation since the overhead is amortized over the time to execute the instruction window. We test window sizes with as few as 16 and as many as 512 instructions.

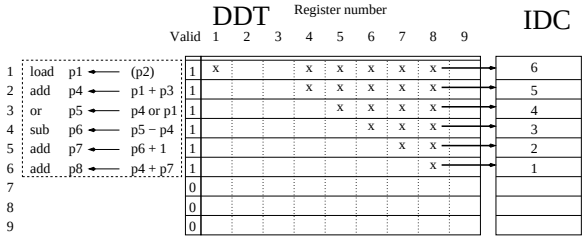
To select the best pipeline for each instruction group, we profile the application on each of the heterogeneous clusters individually and record the performance to execute each window of *committed* instructions on each pipeline. During the actual run, the recorded performance numbers are compared for each window and instructions are steered to the cluster providing the best performance during profiling. This method is imperfect because toggling the steering between clusters affects instruction timing. Perturbing the timing affects branch prediction history updates and how much speculative execution is performed. However, this technique removes to a large degree the uncertainty in how to steer so the results reflect much of the potential of the heterogeneous architecture.

Our second method utilizes the dynamic Data Dependence Tracking (DDT) mechanism proposed in [5]. As shown in Figure 3, the IDC (Instruction Dependences Counter) filter counts the number of instructions dependent on each instruction. This is an important ILP metric. Figure 3(a) shows the IDC values for a group of highly data-dependent instructions. A narrow, fast pipeline is more suitable for this workload with low ILP. By contrast, Figure 3(b) shows a group of instructions with high ILP. A wide pipeline is necessary to exploit this high ILP. As a measure of ILP, we calculate the average IDC value of a given group of instructions. For Figure 3(a), the IDC value average is 3.5 while it is 1.8 for the instructions in Figure 3(b). Thus, the larger the average IDC value, the lower the ILP. Therefore, it is more suitable to use the narrow but fast pipeline for instructions with a large average IDC value. Figure 3(c) shows another group of instructions. The ILP of this group of instructions is larger than those in Figure 3(a) but smaller than the group in Figure 3(b). As a result, the performance difference between running this group of instructions on a wide pipeline and on a narrow pipeline is small. In other words, the group of instructions in Figure 3(c) has less performance sensitivity for the wide or narrow pipeline, compared with the groups in Figures 3(b) and 3(a).

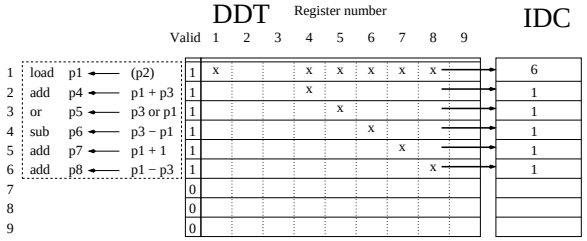
However, the IDC value average of Figure 3(c) (1.7) is almost the same as Figure 3(b) (1.8). We introduce the average "deviation" of the IDC values, calculated as:

$$Deviation_{avg} = \frac{1}{Size_{IDC}} * \sum_{i=a}^{b} |IDC_i - IDC_{avg}|,$$

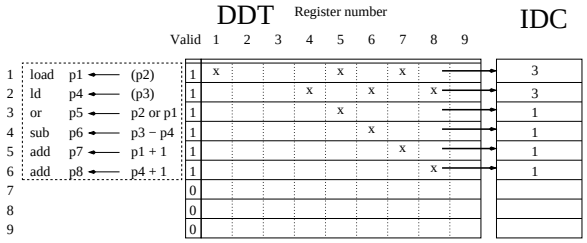
where *a* and *b* are the instruction boundaries of the valid IDC region.



(a) IDC values for a group of instructions with low ILP.



(b) IDC values for a group of instructions with high ILP.



(c) IDC values for a group of instructions with small IDC value average and small deviation average.

Figure 3. Dynamic Data Dependence Tracking (DDT) and Instruction Dependences Counter (IDC) filter.

Therefore, the average deviation for Figure 3(b) is:

$$(6 - 1.8 + (1.8 - 1) * 5)/6 = 1.4.$$

The average deviation for Figure 3(c) is:

$$((3 - 1.7) * 2 + (1.7 - 1) * 4)/6 = 0.9.$$

These results show that for similar average IDC values, the larger the average deviation, the larger the ILP.

The DDT and IDC are updated on a cycle-by-cycle basis. When each instruction commits, the associated DDT and IDC entries are released. In order to calculate the deviation, we maintain the IDC values in an *IDC value buffer* as shown in Figure 4. The buffer size is the same as the steering instruction window size. After the average IDC value is produced, each IDC value deviation is obtained using the average IDC value and each IDC value. Since we choose

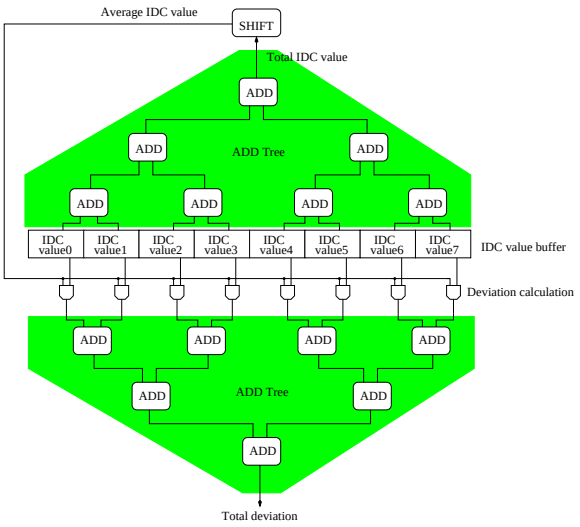


Figure 4. Calculating the average IDC value and deviation.

the steering instruction window to be a power of two, the average can be obtained using a simple shift right operation. Note that instead of calculating the average deviation, the total deviation can be used to compare against the threshold, which can also be presented as the total deviation (average deviation times the steering instruction window size). We estimate the above steps take four cycles. According to our simulation results, this delay does not have much impact on system performance. It is totally hidden if the instruction steering result for the next instruction window is the same as the current one.

3 Methodology

Our simulation environment is based on the SimpleScalar toolset [2]. We have modified the simulator to support clustering of the integer ALU units with different clock frequencies for each cluster. The DDT and the IDC filter are added to collect the dependence information on a cycle-by-cycle basis. A summary of our simulation parameters appears in Table 1.

In the experiments, our baseline is a pipeline having eight integer ALUs. We compare this baseline to a six ALU design at the same frequency; a two ALU design with the issue and execution stages running at twice the baseline frequency; a homogeneous clustered microarchitecture with two identical four ALU clusters; and a heterogeneous cluster architecture having a six ALU cluster at the baseline frequency and a two ALU cluster at the 2X frequency. The instruction steering mechanism for the homogeneous clusters is based on load balancing and instruction criticality as described in [1].

For the heterogeneous cluster architecture, we simulated two instruction steering mechanisms. The first mechanism is based on the IPC profile information of the 6-way and 2-way pipelines. The instruction stream is grouped into

Table 1. Architectural parameters for simulated processor.

Parameter	Value
Fetch Queue	16 entries
Branch predictor	2Bc-gskew
BTB	512 sets, 4-way
Branch Miss Penalty	7 cycles
Decode Width	8 instructions
Issue Width	8 instructions
Retire Width	16 instructions
L1 D/I-Cache	64KB, 4-way set associative
L2 Unifi ed Cache	512KB, 4-way set associative
L1 D/I-Cache Latency	1 cycles
L2 Cache Latency	6 cycles
Memory Latency	60 cycles(fi rst), 2 cycles(subsequent)
FP ALUs	4 + 1 mult/div/sqrt unit
Issue Queue	32 entries
Load/Store Queue	32 entries
Reorder Buffer	256 entries

windows of 16 committed instructions each of which are directed to one of the two clusters in the heterogeneous design.

The second instruction steering mechanism is based on the DDT and the IDC filter as discussed earlier. The average IDC value and deviation are calculated at run time for each group of 16 committed instructions. Note that the tail end execution of a window may overlap with the start up of the next window running on a different cluster. A 1-cycle communication cost is assumed when one cluster needs a value from the other cluster.

The steering decision of the next 16 committed instructions is made by comparing the calculated values against the pre-selected thresholds.

We select a mix of applications from the MediaBench and Spec2000 benchmark suites. Tables 2 and 3 specify the benchmarks along with the input data set and simulation window for the simulation runs.

4 Results

Figures 5 and 6 show the normalized IPC with different architectures.

The difference between the IPC-profile based and the DDT-IDC based instruction steering mechanisms is clearly demonstrated through the examples of *adpcm* and *adpcm2*. The relative performance of the IPC-profile based scheme (1.06 and 1.03, respectively) is worse than using the 2-way pipeline alone for these applications (1.13 and 1.07, respectively). By contrast, the DDT-IDC scheme measures the dynamic average IDC value and deviation and selects the 2-way pipeline for most of the time. The performance is similar to that of using the 2-way pipeline alone.

The performance potential of the heterogeneous architecture is best demonstrated by *gsm2*. We collected the number of cycles spent on each 16-instruction window for the 6-way and 2-way monolithic architectures. We then

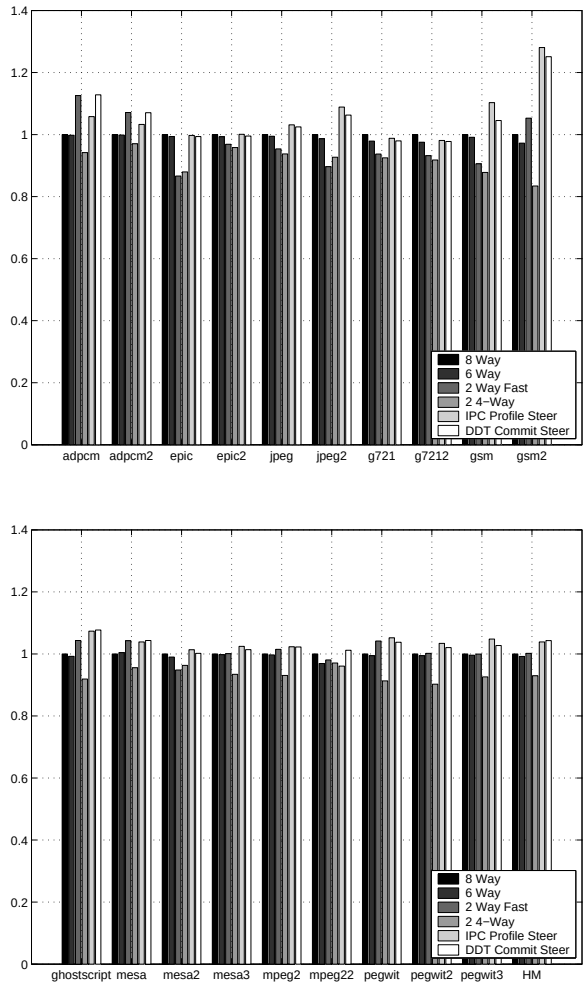


Figure 5. Normalized IPC for MediaBench.

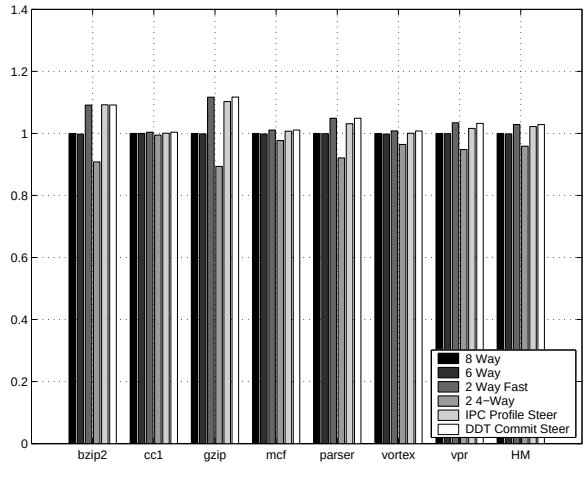


Figure 6. Normalized IPC for SPEC2000.

Table 2. MediaBench benchmark applications.

Benchmark	Datasets	Simulation window
adpcm	ref	encode (6.6M)
		decode (5.5M)
epic	ref	encode (53M)
		decode (6.7M)
jpeg	ref	compress (15.5M)
		decompress (4.6M)
g721	ref	encode (0–200M)
		decode (0–200M)
gsm	ref	encode (0–200M)
		decode (0–74M)
ghostscript	ref	0–200M
mesa	ref	mipmap (44.7M)
		osdemo (7.6M)
		texgen (75.8M)
mpeg2	ref	encode (0–200M)
		decode (0–171M)
pegwit	ref	encrypt key (12.3M)
		encrypt (32.4M)
		decrypt (17.7M)

Table 3. Spec2000 Integer Benchmark applications.

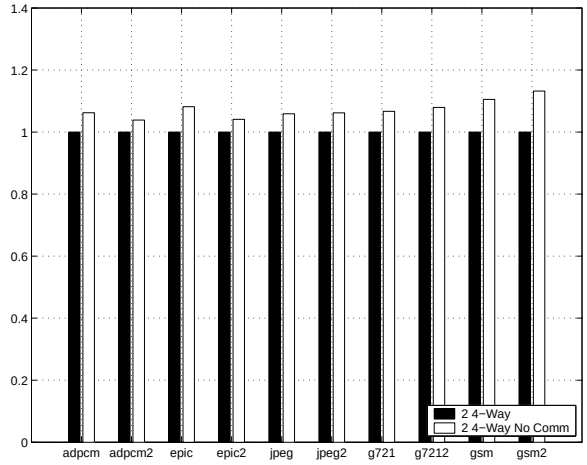
Benchmark	Datasets	Simulation window
bzip2	source 58	1000M–1100M
gcc	166.i	2000M–2100M
gzip	source 60	1000M–1100M
mcf	ref	1000M–1100M
parser	ref	1000M–1100M
vortex	ref	1000M–1100M
vpr	ref	1000M–1100M

combined the smaller number of cycles among the two for each instruction window and calculated the "combined" IPC. The result is a 29.7% improvement over the 8-way machine, which is closely matched by the heterogeneous approach. Using the heterogeneous clustered architecture effectively reduces the fraction of the time when the fetch queue is full (from 46.1%, 46.7%, and 37.7% for the 8-way, 6-way, and 2-way superscalar, to 16.4% for the heterogeneous cluster).

Overall, the DDT-IDC instruction steering scheme achieves a performance improvement of 4.3%, slightly better than the improvement of 3.9% from the IPC-profile based instruction steering mechanism.

The homogeneous cluster of two 4-way pipelines does not perform very well mainly due to the communication cost. Figures 7 and 8 show the performance differences between the same homogeneous clustered architecture with and without a 1-cycle communication cost for different applications. The abnormal case of *mpeg22* is due to the branch predictor performance change (prediction accuracy decreased from 95.6% to 93.3% when removing the communication cost). The block based instruction steering schemes effectively remove most communication cost.

Figure 9 shows the utilization of the 6-way and 2-way



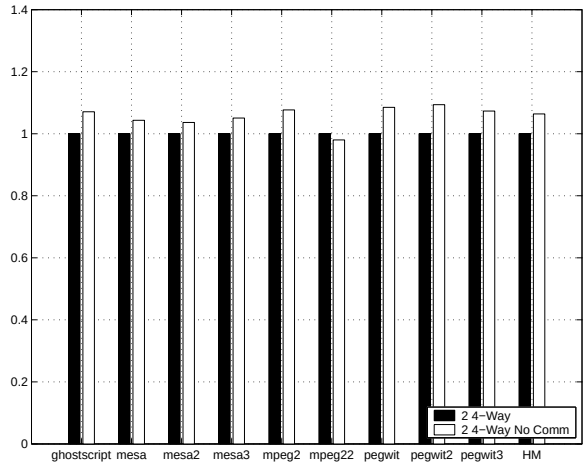


Figure 7. Homogeneous cluster performance difference for MediaBench.

clusters using different steering mechanisms. For the SPEC 2K integer applications shown in Figure 6, the 6-way cluster shows the same performance as the baseline 8-way architecture, which reflects the relatively low ILP of these benchmarks. However, this low ILP can be exploited by the 2-way fast cluster. Using the 2-way cluster alone achieves the best overall performance (2.9% performance improvement over the baseline). The DDT-IDC based instruction steering scheme uses the 2-way cluster most of the time, resulting in similar performance as the 2-way cluster (2.8% performance improvement over the baseline). The IPC-profile based steering scheme performs slightly worse (2.2% over the baseline) due to its more frequent use of the 6-way cluster. In summary, the DDT can be used to derive an effective steering mechanism for heterogeneous clustered microarchitectures, with an overall performance improvement of 4.3% over a variety of integer benchmarks.

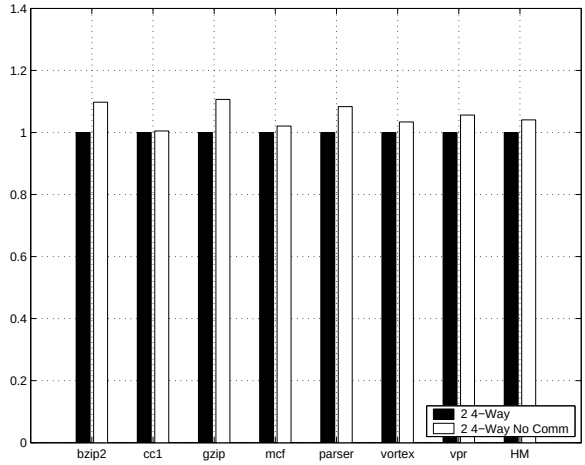


Figure 8. Homogeneous cluster performance difference for SPEC2000.

5 Related Work

There has been a significant body of work in *homogeneous* clustered microarchitectures. We summarize those efforts that are most related to our heterogeneous design.

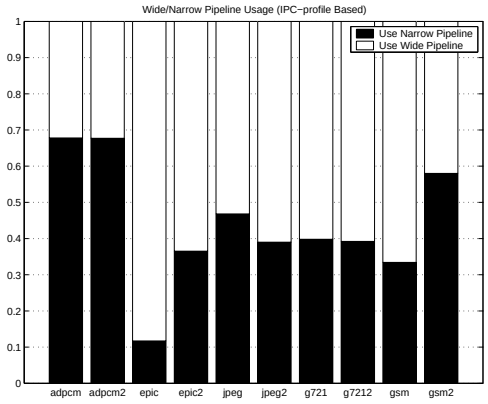
The multicluster architecture [6] distributes the register files, dispatch queue, and functional units to multiple clusters. It applies a static instruction scheduling algorithm to assign instructions to each cluster. A conventional superscalar processor can be considered as two clusters, i.e., an integer cluster and a floating-point cluster. Canal *et al.* propose several run-time mechanisms to distribute simple integer instructions to both integer and FP clusters [3]. In later work, Canal *et al.* propose Simple Register Mapping Based Steering (RMBS), Balanced RMBS, and Advanced RMBS for instruction steering between two clusters [4].

Parcerisa *et al.* propose point-to-point interconnects for clustered microarchitectures [11]. A topology-aware steering heuristic is designed to minimize communication cost while keeping cluster load balanced.

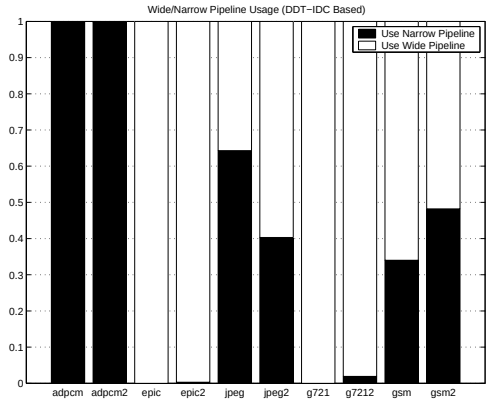
Ranganathan *et al.* categorize the decentralized execution models to *execution unit dependence based*, *control dependence based*, and *data dependence based* [12]. It is shown that with a ring-type interconnect, data dependence based decentralization performs the best with moderate number of processing elements. When the number of processing elements gets larger, the control dependence based decentralization performs better.

Balasubramonian *et al.* apply dynamic techniques to tune the clustered architecture to better suit application needs [1]. The techniques are based on program metrics and gathered at periodic intervals. It is also shown that reconfiguration at basic block boundaries can further improve system performance.

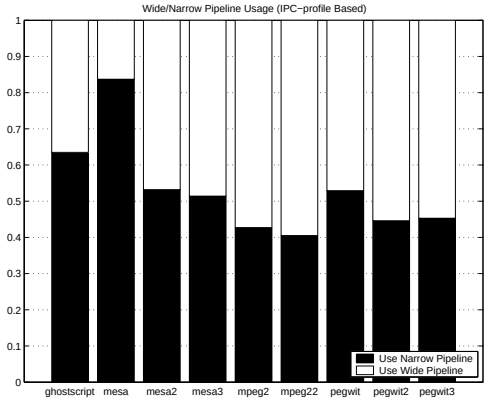
Kumar *et al.* integrate heterogeneous processor *cores* in [9, 10]. The cores are complete designs of the Alpha family of processors including L1 data cache. The cores range



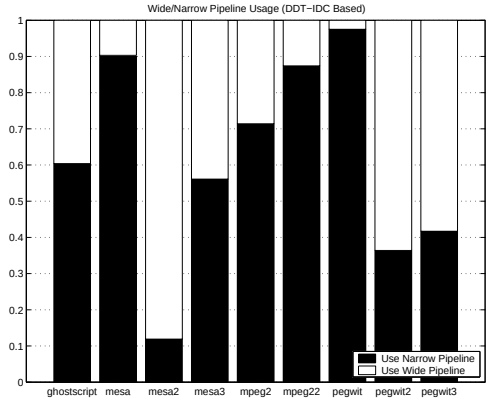
(a) MediaBench 1, IPC-profi le based steering



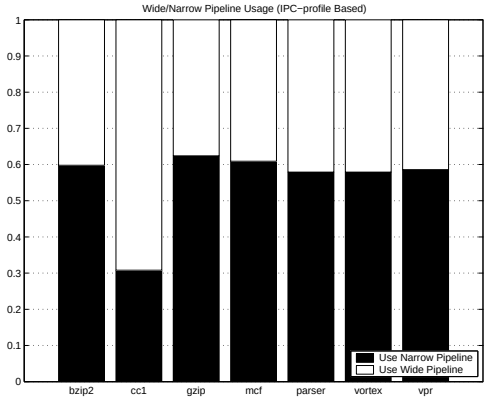
(b) MediaBench 1, DDT-IDC based steering



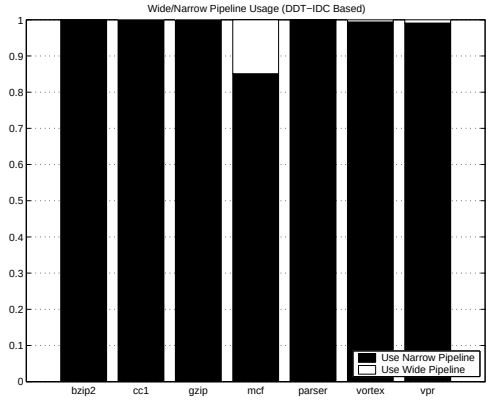
(c) MediaBench 2, IPC-profi le based steering



(d) MediaBench 2, DDT-IDC based steering



(e) SPEC2K, IPC-profi le based steering



(f) SPEC2K, DDT-IDC based steering

Figure 9. Wide versus narrow core usage breakdown.

from simple, low power embedded designs to the EV8 design. In [9], the work explores reducing energy by matching applications to the simplest core (and lowest energy) that offers sufficient performance. In [10], a multi-core architecture is used to improve the performance of multi-threaded workloads. Because the caches are included with each core, switching cores necessitates flushing the active cache of dirty data. This overhead restricts the steering to be very coarse-grained. Our proposed design, in contrast, targets performance and relies on the ability to make fine-grained steering decisions in order to exploit differences in ILP in sections of the code.

6 Conclusion

The static microarchitecture of a conventional superscalar processor microarchitecture may be poorly matched to the ILP characteristics of the application at hand. The result may be lower performance than can be achieved on a pipeline that achieves high throughput for narrow streams of instructions. We propose a heterogeneous clustered microarchitecture that attempts to bridge the gap between these two design styles. In our design, windows of instructions are steered to either a wide pipeline optimized for high ILP or a narrower pipeline optimized for narrow streams of instructions. This steering approach largely hides the inter-cluster communication penalties of prior, homogeneous, clustered architectures. The result is up to a 30% performance improvement compared to a conventional approach.

This paper represents an initial step in our research in this area. For future work, we plan to investigate the dynamic choice of instruction window sizes, and mechanisms to identify code regions where simultaneous use of both the wide and narrow pipelines would yield additional performance benefits.

References

[1] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically Managing the Communication-Parallelism Trade-off in Future Clustered Processors. *Proceedings of the 30th International Symposium on Computer Architecture*, pages 275–286, 2003.

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

[3] R. Canal, J. M. Parcerisa, and A. Gonzalez. Dynamic Cluster Assignment Mechanisms. *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, pages 133–142, 2000.

[4] R. Canal, J. M. Parcerisa, and A. Gonzalez. Dynamic Code Partitioning for Clustered Architectures. *International Journal of Parallel Programming*, pages 59–79, 2001.

[5] L. Chen, S. Dropsho, and D. Albonesi. Dynamic data dependence tracking and its application to branch prediction. In *Ninth International Symposium on High-Performance Computer Architecture*, pages 65–76, Feb. 2003.

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

[7] M. Goshima, K. Nishino, Y. Nakashima, S. ichiro Mori, T. Kitamura, and S. Tomita. A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors. *IPSJ Transactions on High Performance Computing Systems*, pages 225–236, December 2001.

[8] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 Microprocessor Architecture. *1998 International Conference on Computer Design*, pages 24–36, October 1998.

[9] R. Kumar, K. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Processor Power Reduction via Single-ISA Heterogeneous Multi-Core Architectures. *IEEE Architectural Letters*, March 2003.

[10] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. *Proceedings of the 31st International Symposium on Computer Architecture*, 2004.

[11] J. M. Parcerisa, J. Sahuquillo, A. Gonzalez, and J. Duato. Efficient Interconnects for Clustered Microarchitectures. *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 291–300, 2002.

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