A Software-Hardware Hybrid Steering Mechanism for Clustered Microarchitectures

Qiong Cai Josep M. Codina José González
Antonio González
Intel Barcelona Research Centers, Intel-UPC
{qiongx.cai, josep.m.codina, pepe.gonzalez, antonio.gonzalez}@intel.com

Abstract

Clustered microarchitectures provide a promising paradigm to solve or alleviate the problems of increasing microprocessor complexity and wire delays. High-performance out-of-order processors rely on hardware-only steering mechanisms to achieve balanced workload distribution among clusters. However, the additional steering logic results in a significant increase on complexity, which actually decreases the benefits of the clustered design.

In this paper, we address this complexity issue and present a novel software-hardware hybrid steering mechanism for out-of-order processors. The proposed software-hardware cooperative scheme makes use of the concept of virtual clusters. Instructions are distributed to virtual clusters at compile time using static properties of the program such as data dependences. Then, at runtime, virtual clusters are mapped into physical clusters by considering workload information.

Experiments using SPEC CPU2000 benchmarks show that our hybrid approach can achieve almost the same performance as a state-of-the-art hardware-only steering scheme, while requiring low hardware complexity. In addition, the proposed mechanism outperforms state-of-the-art software-only steering mechanisms by 5% *and* 10% *on average for 2-cluster and 4-cluster machines, respectively.*

1. Introduction

Clustered microarchitecture design is attractive for the next generation of microprocessor technology, because it circumvents many power, thermal and complexity problems faced by today's computer architects [7, 14]. A clustered microarchitecture distributes processor resources into several partitions or clusters. The components of each cluster are simpler, faster and much more power-efficient than its monolithic counterparts.

978-1-4244-1694-3/08/$25.00 ©2008 IEEE

In a clustered microarchitecture, a value produced by one cluster and consumed by another cluster needs to pay extra communication cost. The simplest way to achieve zero communication cost is to send all dependent instructions into one cluster. However, this naive method yields the worst workload distribution. The main challenge for a clustered microarchitecture is the design and implementation of a *steering unit*, which is responsible for distributing instructions among clusters so that the communication cost is minimized and the workload is balanced.

A number of high-performance hardware-only steering mechanisms have been already proposed [3, 4, 15] to mitigate most of the penalties introduced by the communications. However, they are fairly complex to be implemented in a real processor due to tight timing constraints of the pipeline. In particular, the complexity of the steering logic is much higher than that of the register renaming. Our objective is to reduce such complexity and maintain the performance achieved by the hardware-only schemes.

Different software-only steering techniques have been proposed [14, 19, 26] and they are particularly common in statically-scheduled processors (e.g. VLIW) [6, 13, 8, 12, 16, 17, 21], where the compiler is responsible for both code scheduling and instruction distribution among clusters. However, as we will show later in this paper, the software-only approach performs much worse than its hardware-only counterpart when it is applied to out-of-order processors.

To reduce the complexity of the hardware-only approach and improve the performance of the software-only approach, we propose a software-hardware hybrid steering mechanism for clustered x86 out-of-order processors. The proposed technique introduces the concept of *virtual clusters* [1] to make software and hardware work together. The instructions are partitioned into virtual clusters at compile time by considering static properties of a program such as

[1] The concept of virtual clusters and the way they are formed as proposed in this paper are completely different from [11], where the virtual cluster is used as an interface to combine cluster assignment and instruction scheduling in a single step for VLIW processors.

data dependences. At execution time, the hardware utilizes runtime workload information to adjust the decisions of instruction distribution made at compile time and maps virtual clusters into physical ones.

Performance results show that the proposed software-hardware hybrid steering mechanism achieves performance close to a state-of-the-art hardware-only steering algorithm [15] with an average slowdown lower than 2%. We also show that our approach outperforms two state-of-the-art software-only steering algorithms by almost 5% and 10% on average for 2-cluster and 4-cluster processors, respectively.

The main contributions of this paper can be summarized as follows:

1. A novel software-hardware hybrid steering algorithm is proposed by using virtual clusters. A virtual cluster serves as an interface between software and hardware so that our algorithm can take advantages of both sides. By using virtual clusters, our mechanism removes most of the steering complexity in the hardware-only approach.

2. A detailed quantitative analysis is provided comparing software-only, hardware-only and our hybrid approaches. This comparison reveals that our hybrid mechanism achieves performance close to the state-of-the-art hardware-only steering mechanism and outperforms the state-of-the-art software-only mechanism.

The rest of the paper is organized as follows. In Section 2, we describe our baseline: a clustered x86 out-of-order microarchitecture. In particular, we explain why the complexity of a steering unit is a real issue in clustered microarchitectures. In Section 3, we review the previous work on steering for clustered microarchitectures including hardware-only and software-only approaches. After seeing the advantages and disadvantages of both approaches, we propose our software-hardware hybrid approach in Section 4. The performance results on different approaches are shown in Section 5. Finally, we conclude in Section 6.

2. The Clustered Microarchitecture

Figure 1 shows our baseline clustered x86 out-of-order microarchitecture design. This architecture consists of a monolithic frontend and a clustered backend. In the frontend, the dispatching unit receives micro-ops coming from the instruction cache, and the steering logic is responsible for sending these micro-ops to the appropriate clusters according to a particular steering policy. All backend clusters have their own register files, issue queues, integer functional units and floating point functional units. Moreover, the

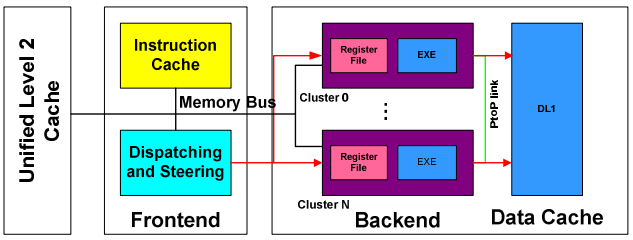

Figure 1. Block diagram of the clustered microarchitecture

backend clusters are connected through a dedicated point-to-point interconnection network.

The Load/Store Queue (LSQ) and the data cache are unified and accessed by clusters through dedicated buses. At dispatch time, loads and stores reserve a slot in LSQ and they are steered to the corresponding cluster, where the effective address is computed. Memory operations are stored in the LSQ, and remain there until they access the data cache.

In this clustered processor, once a micro-op is steered to a particular cluster, say cluster A, it remains in the issue queue until its input operands become available. When all operands are ready, the instruction leaves the issue queue and proceeds to the execution units. However, if any required operand is generated from a different cluster, say cluster B, a communication is required to transfer the value from cluster B to cluster A. In order to perform this communication, an explicit ***copy micro-op*** must be generated and inserted in cluster B. This communication instruction will be responsible for transferring the value generated in cluster B to cluster A through a point-to-point link.

2.1. Complexity Issue of Steering Unit

The complexity issue of a steering unit has not been well addressed in the literature. The previous hardware-only steering mechanisms did not pay particular attention to the tight timing constraints of the pipeline. High performance steering mechanisms based on the characteristics of the dependences are much more complex than the register renaming.

In register renaming, all inputs and outputs are first renamed in parallel, and then an overriding phase updates intra-dependences in the decode bundle. However, the dependence-based steering is different. An instruction is distributed to a cluster holding most of its inputs. In case of a tie, it is sent to the least loaded cluster. If steering is implemented as register renaming, all instructions in the decode bundle would check the locations of their inputs in parallel, and then the destination cluster of each instruction would be decided in parallel through a voting process. The vot-

ing process uses non-updated information, which results in significant performance degradation. For example, we have three instructions to be distributed:

I1: R1 ← R1 + R2

I2: R3 ← Load (R1)

I3: R4 ← Load (R3)

We assume that before steering, R1 was in cluster 0, R2 and R3 were in cluster 1, and cluster 1 is empty. If we use *parallel steering*, which is similar to register renaming, we would have the following scenario:

1. I1 goes to cluster 1 (there is a tie and cluster 1 is empty).
2. I2 goes to cluster 0 (R1 was located in cluster 0).
3. I3 goes to cluster 1 (R3 was located in cluster 1).

This will generate two copies. To have better steering results, a *sequential steering* must be implemented. In this case:

1. I1 goes to cluster 1 as usual.
2. I2 knows that R1 is in cluster 1 now and it is send to cluster 1.
3. I3 is also sent to cluster 1.

Therefore, we have zero copies. To have fewer copy instructions, the hardware-only steering should be implemented in a sequential fashion. However, this increases complexity dramatically and, may not met the cycle time requirements when it is implemented under tight timing constraints. In order to overcome this complexity, our software-hardware hybrid mechanism removes most of the hardware-only steering logic and maintains the same performance as its hardware-only counterpart.

3. Steering Mechanisms

One of key challenges for clustered microarchitectures is to properly distribute instructions among clusters. Several schemes have been proposed in the literature to deal with this problem. In spite of the fact that they have the common goal of achieving the workload balance and minimizing the communication cost at the same time, they are different in nature. In particular, previous work can be divided into three main categories: (i) hardware-only mechanisms commonly applied to dynamically-scheduled (or out-of-order) processors, (ii) software-only mechanisms applied to dynamically-scheduled processors, and (iii) software-only mechanisms applied to statically-scheduled processors such as VLIW.

In this section, we review previous work according to the above classification. For each category, one state-of-the art technique is selected and discussed work in details. The selected mechanisms are implemented in our simulation infrastructure and compared with our hybrid mechanism in Section 5.

3.1. Hardware-based Mechanisms

Most dynamically-scheduled clustered microarchitectures rely on dynamic steering of instructions. Several hardware-only steering mechanisms have been proposed [3, 5, 15, 24]. Each one of them uses different heuristics to perform instruction distribution among clusters by exploiting runtime information. Most of the hardware-only steering policies consider data dependences and cluster occupancies when steering a particular instruction. Some recent work has pointed out the benefit of *stalling over steering* [15, 24]. The rationale behind that is in some cases it is better to stall the processor frontend, rather than steering instructions to the less loaded cluster regardless the location of the operands, which may generate fewer copy instructions in the critical path. The work presented in [24] shows a theoretical study of criticality and clustering, presenting (with no particular implementation) several enhancements to current steering policies. In [15], an *occupancy-aware steering* is presented. The practical implementation of this technique stalls the steering unit if the preferred cluster cannot be chosen (due to lack of resources) and the other ones are busy. In this paper, the occupancy-aware policy will be our baseline for comparison purposes. Although the hardware-based mechanisms can achieve high-performance, the additional complexity added to the hardware puts timing constraints on the implementation of these schemes.

3.2. Software-based Mechanisms for Dynamically-scheduled Processors

A very limited number of previous studies have proposed software-only mechanisms for dynamically-scheduled processors [14, 19, 26]. There are two advantages of using a software-only mechanism for clustered out-of-order processors. First, a bigger window of instructions is inspected at compile time in order to make steering decision, which may potentially reduce the number copy instructions generated. Second, it completely eliminates the dynamic steering complexity.

Nagarajan et al [19] presented an execution model called static placement dynamic issue (SPDI) for Explicit Data Graph Execution (EDGE) architectures [4]. The EDGE architecture has similarities to a clustered out-of-order design. In particular, in an EDGE architecture several ALUs (or

clusters) are connected through an on-chip network interconnects (e.g. a mesh). On top of this architecture, the SPDI execution model relies on the compiler to map the instructions into different ALUs and allows the hardware to dynamically issue the instructions once their operands are available. Our software-hardware hybrid approach also relies on the compiler to map instructions to different clusters and allows the hardware to issue the instructions dynamically. The main difference between these two approaches is that the mapping decision made by the compiler is refined at runtime in our software-hardware hybrid scheme. We will show later in Section 5 that our proposed software-hardware approach significantly outperforms the scheme used in SPDI.

3.3. Software-based Mechanisms for Statically-scheduled Processors

The static steering mechanism is a common approach for statically-scheduled clustered processors (e.g. VLIW). All previous work can be classified according to their compilation scopes. One important category is the code generation for loops [1, 2, 10, 20, 25] by means of modulo scheduling techniques [9, 23]. Another category schedules instructions for more general program structures including cyclic and acyclic control flow graphs [6, 8, 12, 13, 16, 17, 21]. In this paper, we focus on the latter category.

One of the state-of-the-art algorithms is based on a formulation of the cluster assignment problem into a graph partitioning problem, solved by a multilevel graph partitioning algorithm [1, 2, 8]. A multi-level graph partition algorithm [18] consists of two steps: coarsening and refinement. The coarsening step produces an initial partition for the graph and the refinement step iteratively refines the initial partition based on heuristics. RHOP [8] is an example of multilevel graph partitioning algorithm applied to the cluster assignment problem. In RHOP, the weights are assigned to nodes and edges in the data dependence graphs based on slack information computed from the static latencies of the instructions. The coarsening stage in RHOP tends to group the operations on the critical path together and it stops coarsening instructions when the number of coarse nodes equals the number of clusters in the machine. The refinement stage traverses back through the coarsening step and make improvement to the initial partition based on the metrics such as the workload per cluster and total system workload. The aim is to balance the workload and minimize the inter-cluster communications.

RHOP was originally defined for VLIW processors and the algorithm relies on the accuracy of the estimated workload at compile time. However, it is much harder to estimate the workload for out-of-order machine. The software-hardware hybrid scheme proposed in this paper addresses this issue by a co-designed effort. Results reported in this paper shown that this hybrid algorithm significantly outperforms RHOP when both are applied to a dynamically-scheduled processor.

4. The Software-Hardware Hybrid Steering: Virtual Cluster

This section presents our software-hardware hybrid steering scheme for a clustered out-of-order microarchitecture. In the process of designing our algorithm, the advantages and disadvantages of software-only and hardware-only schemes have been taken into account. In particular, software-only approaches have better view of data dependences but cannot accurately estimate runtime information such as workload balance, as discussed in the previous section. On the other hand, hardware-only approaches have much more accurate knowledge of runtime information but have very limited view of data dependences. These observations lead us to design a software-hardware hybrid steering algorithm.

4.1. Virtual Clustering: A bridge from software to hardware

To make software and hardware work in harmony and achieve a cost-effective solution for the steering problem, we propose the concept of virtual clustering. The virtual cluster is an interface between software and hardware, and it is managed by both the compiler and the hardware steering mechanism. The compiler is in charge of assigning instructions to virtual clusters, while a simple hardware steering logic performs the mapping from virtual clusters to physical clusters. The number of virtual clusters is fixed by the hardware and exposed to the software through the instruction set.

By using the virtual cluster model, the proposed hybrid scheme can combine the benefits from both software-only and hardware-only approaches. The compiler can build larger data dependence graphs than the hardware does, and the hardware can refine the initial steering decisions made by the compiler based on accurate runtime workload information.

In the next sections, we describe in details the two main steps of our proposal: the partitioning of data dependence graph (DDG) into virtual clusters and the mapping of virtual clusters into physical clusters.

4.2. Software Partitioning into Virtual Clusters

In Figure 2, the algorithm for partitioning a data dependence graph (DDG) into virtual clusters at compile time is

```
Partitioning the DDG into virtual clusters (DDG G)
{
  1. compute_critical_info (G)
  2. partition_ddg_based_on_critical_info (G)
  3. FOREACH_DDG_NODE (G, node)
        same_chain_p = False;
        FOREACH_DDG_PREDECESSOR(node, pred)
            if ( node->v_cluster == pred->v_cluster)
                same_chaip_p = True;
                break;
        if (same_chain_p)
            node->chain_leader = 0;
        else
            node->chain_leader = 1;
}
```

Figure 2. Partition DDG nodes into virtual clusters at compile time

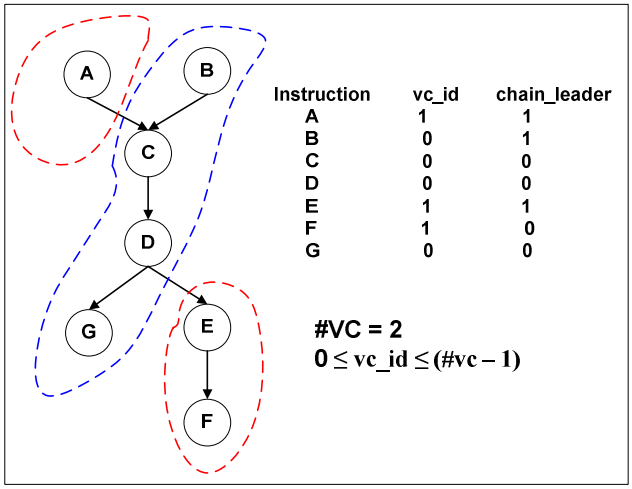

Figure 3. Chain Leader

shown. This algorithm is divided into three main steps:

Computation of critical paths. For a given DDG, the compiler first computes the critical path information. This computation requires two traversals of a DDG: one for computing the depth and another for computing the height of each node in the DDG [19]. The criticality of each node in the DDG is then defined to be the sum of its depth and height. Based on the criticality, critical paths in the DDG are found.

Partition of DDG into virtual clusters. Nodes are partitioned into virtual clusters (VC for short), according to different critical paths. The proposed algorithm traverses the DDG top-down, assigning at each step one instruction to a VC. The algorithm attempts to include all dependent instructions in the same VC. It takes into account the criticality of the instructions when distributing them among VCs. In particular, for each instruction, the benefit of assigning the instruction to all possible VCs is computed and the cluster with the best benefit is selected. In order to compute such expected benefit, the completion time of the instruction is used. In the proposed scheme, the completion time for a particular instruction is estimated based on the dependences, the latencies, and the resource contention in the intended cluster.

Note that the estimation may not be accurate enough for a dynamically-scheduled processor, as discussed in Section 3.2, due to the lack of dynamic information. However, our software-hardware hybrid mechanism can alleviate this problem by performing the final mapping from virtual clusters to physical clusters at runtime based on the dynamic workload information.

Identification of chains and chain leads. The final step involves assignments of identifiers to virtual clusters (`vc_id`) and identifications of *chains* and *chain leaders*. In our algorithm, we refer to a group of instructions in the same virtual cluster that are mapped into the same physical cluster as chains. The chain leader is defined as the first instruction of a chain. Special codes are generated for chain leaders in order to notify the hardware when to update the mapping table between virtual clusters and physical clusters.

The example in Figure 3 shows how virtual clusters and chains are managed. The DDG in this example is partitioned into two virtual clusters. Each node in the DDG is identified by a `vc_id`. The chain leaders are nodes A, B and E. Non chain-leaders are marked with zero.

The effectiveness of our proposed hardware-software mechanism largely depends on the selection of chains. The chain leaders are places in which the hardware checks the runtime workload balance and maps a whole chain into a physical cluster.

4.3. Hardware Mapping Virtual Clusters into Physical Clusters

The steering logic required by our approach is much simpler than the steering logic in a traditional clustered processor. Specifically, the steering unit in our hybrid approach is only responsible for performing the appropriate mapping between virtual clusters generated at compile time and physical clusters. The only hardware required is: (1) a set of counters that indicates the distribution of instruc-

```
Mapping virtual clusters into physical clusters(uop s)
{
  vc_id = get_virutal_clusters ( s->ip );
  chain_leader = is_chain_leader ( s->ip );
  if ( chain_leader == 1 )
        p_cluster = the least_loaded_cluster ();
        update_virtual_cluster_table (vc_id, p_cluster)
  else
        p_cluster = get_physical_cluster (vc_id);
}
```

Figure 4. Mapping virtual clusters to physical clusters at run time

tions among clusters; and (2) a small table to keep track of the mapping between virtual clusters and physical clusters. Note that, the number of counters required is equal to the number of physical clusters minus 1, and the number of entries in the mapping table is equal to the number of VCs.

Figure 4 shows the algorithm to map virtual clusters into physical clusters at runtime. When decoding an instruction, if the mark of chain leader is encountered, the workload balance counters are checked. Based on the contents of these counters, the VC is mapped to the less loaded cluster.

When a mapping decision is made, the mapping table is updated by setting the `vc_id` to the selected physical cluster. All non-leader instructions (with the mark bit set to 0) will then follow the mapping decision made for their chain leaders, and they will be sent to the same physical cluster.

Once the destination of a physical cluster is decided, the copy generation step is executed. This task is performed as in the traditional clustered architectures. The copy instructions are generated if any of the input operands is produced in some other cluster. The logic that indicates the location of a register value can be attached to the rename table with a negligible complexity increase.

Table 1 shows the comparison of the complexity between our hybrid approach and the hardware-only approach. Four main components are included in the hardware-only design:

1. The dependence checking is in charge of obtaining the location of source registers. It is implemented by means of a table, accessed with the input register identifier to obtain the location of a source value and with the destination register identifier to store the cluster identifier that will produce that value.

2. The workload balance management consists of a set of counters that store the number of in-flight instructions in each cluster.

3. The vote unit takes into account the location of the inputs of the instruction as well as the workload balance information and decides the destination cluster that minimizes communications and keeps clusters balanced.

4. The copy generator is responsible for generating copy instructions when required.

steering algorithm	hardware-only occupancy-	hybrid virtual clustering
dependence check	yes	no
workload balance management	yes	yes
vote unit	yes	no
copy generator	yes	no

Table 1. Complexity comparison between hardware-only occupancy-aware steering and our hybrid virtual clustering.

When the hybrid scheme is applied, dependence checking and voting unit are removed from the hardware side. These two units are the most expensive parts, both in complexity and delay, of a hardware-only scheme. In order to steer a given instruction precisely, all previous instructions in the decode group must be already steered. This is due to the fact that it is necessary to know the exact location of each input. Hence, the steering logic becomes a serialized task in traditional clustered architectures as discussed in Section 2.1. With our hybrid steering algorithm, the hardware complexity due to the serialization of the steering decision is fully removed.

5. Performance Evaluation

5.1. Simulation Framework

Our software-hardware hybrid steering algorithm consists of two parts. The software part (see Figure 2) is implemented in the code generation step of Intel production compiler for x86 microarchitectures, and the hardware part (see Figure 4) is implemented in an event-driven simulator that executes traces of IA32 binaries. In addition, the x86 instruction set is extended in our simulation framework in order to allow the virtual cluster information to be passed from the compiler to the hardware. Table 2 shows the main architectural parameters of our processor.

The PinPoints tool [22] is used to select representative simulation points for SPEC CPU2000 benchmarks. Every simulation point contains 10 million instructions and the maximum number of phases is set to 10. For most of the benchmarks, there are less than 10 phases. The results pre-

Front-end	
Fetch	24K micro-op trace cache, 6 micro-ops/cycle, 5 cycle fetch-to-dispatch
Decode, rename and steer	3+3 micro-ops/cycle, 1 cycle latency
Reorder Buffer	256+256 entries, commit 3+3 micro-ops/cycle
Back-end (configuration shown per cluster)	
Issue queues	48-entry INT, 2 micro-ops/cycle, 48 entry FP, 2 micro-ops/cycle, 24-entry COPY, 1 micro-ops/cycle
Register file	256-entry INT register file, 256-entry FP register file
Inter-cluster communication	bi-directional point-to-point link, 1 cycle latency, 1 copy/cycle
L1 data cache	32KB, 4-way, 3 cycle hit, 2 read ports, 1 write port, 256-entry Load/Store Queue
Memory	
L2 unified cache	2MB, 16-way, 13 cycle hit, $\geq$ 500 cycle miss, 1 read port, 1 write port

Table 2. The architectural parameters

Configuration	Description
OP	Occupancy-aware steering [15]
one-cluster	Every instruction goes to one cluster
OB	Static-placement dynamic issue operation-based steering [19]
RHOP	Region-based hierarchical operation partition [8]
VC	Our hybrid steering based on virtual clustering

Table 3. Five configurations in the experiment

sented in the section are weighted by the weights generated by PinPoints.

In this paper, the five steering mechanisms shown in Table 3 are evaluated. To demonstrate that our steering algorithm can achieve almost the same performance as its hardware-only counterpart, we choose occupancy-aware steering algorithm (OP)[15] (one of the best hardware-only steering algorithms in the literature) as our baseline algorithm. To see how good the OP algorithm is, a naive hardware-only mechanism called one-cluster, which steers every instruction to the same physical cluster, is also evaluated. In order to further demonstrate the benefits of our approach, we implemented two state-of-the-art software-only algorithms: static-placement dynamic-issue scheduling (OB) [19] and RHOP steering algorithm [8] described in Section 3.

Our proposed hybrid steering algorithm is referred to as VC. The base architecture is a 2-cluster machine. The number of virtual clusters is set to 2, because such configuration achieves almost the same performance as the configurations with the increased number of virtual clusters.

Finally, to demonstrate our algorithm can scale well beyond a 2-cluster machine, we will also show the performance results for a 4-cluster machine in Section 5.4.

5.2. Performance Results

Figure 5 shows the performance results of one-cluster, OB, RHOP and VC schemes with respect to the best hardware-only scheme (i.e., OP). We can see that our hybrid approach outperforms the software-only approach for almost every benchmark. For floating benchmarks, VC can achieve almost 5% average speedup and obtain the performance improvement up to 20% for benchmarks such as galgel. Moreover, the performance of our hybrid approach is very close to the hardware-only approach with only 2.62% slowdown.

5.3. Copy Reduction and Workload Balance Improvement

The aim of a steering mechanism is to minimize the number of copy instructions generated and maximize the workload balance. However, these are two conflicting goals, and achieving them at the same time is a NP problem [1, 8, 18]. To demonstrate the tradeoff between the minimization of copy instructions and the maximization of the workload balance, we compare our hybrid algorithm (VC) with OB, RHOP and OP algorithms in terms of copy reduction and workload balance improvement. In this experiment, workload balance improvement is computed as the total reduction of the allocation stalls in the issue queues.

Figures 6(a.1) and (b.1) show the copy reduction and workload balance improvement of VC over OB with respect to the speedups. Every point in the figure refers to a trace gathered by the PinPoints tool. The x-axis represents the speedup, whereas the y-axis shows the copy reduction and workload balance improvement in figures (a.1) and (b.1), respectively. We can clearly see that VC reduces the number copies and improves the workload balance for most of the traces. This results show the reasons why VC outperforms OB for most of the benchmarks.

Figures 6(a.2) and (b.2) show the same comparisons between VC and RHOP. The main observation that we can

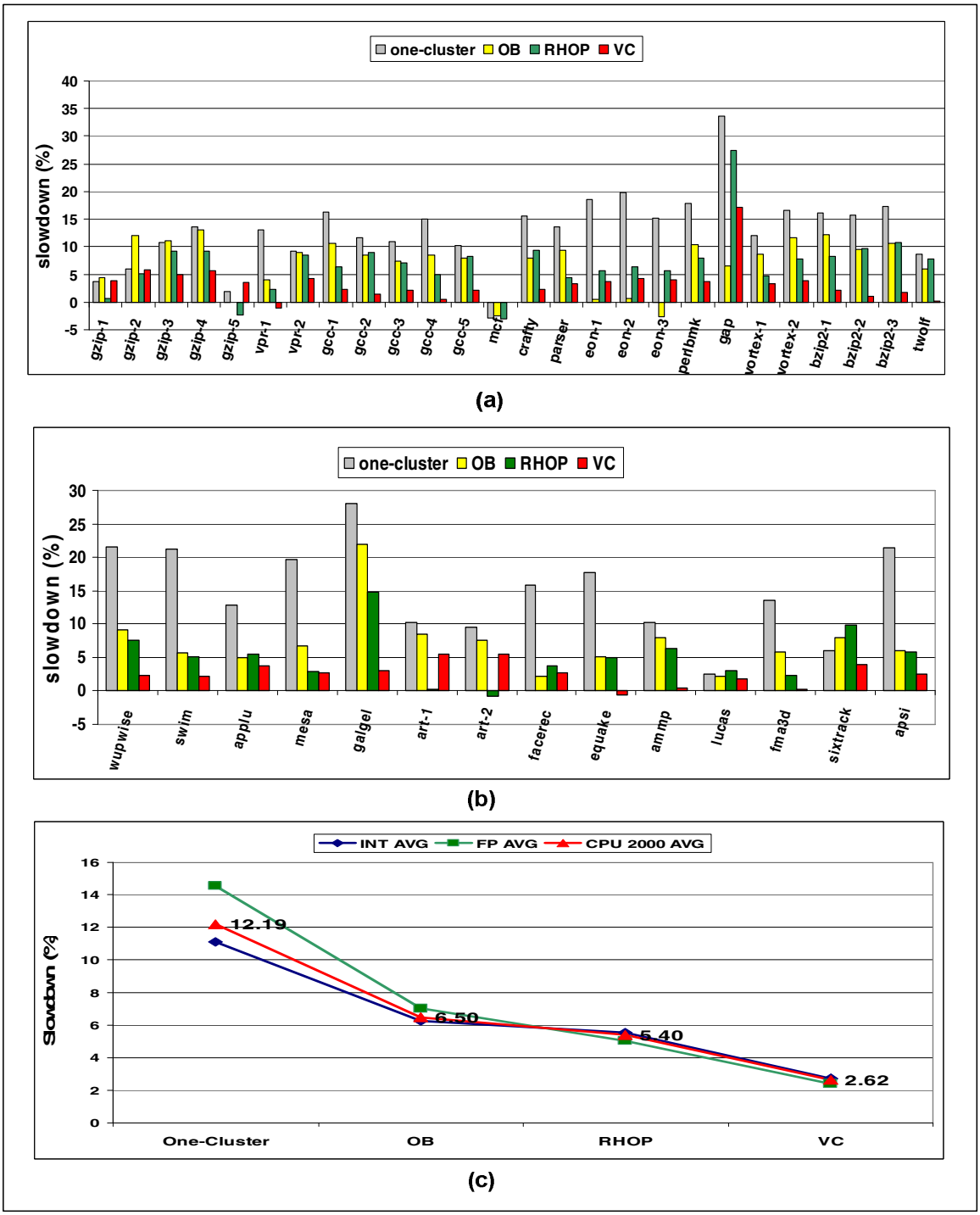

Figure 5. The performance results with respect to configuration OP: (a) SpecInt 2000 (b) SpecFP 2000 (c) the average results.

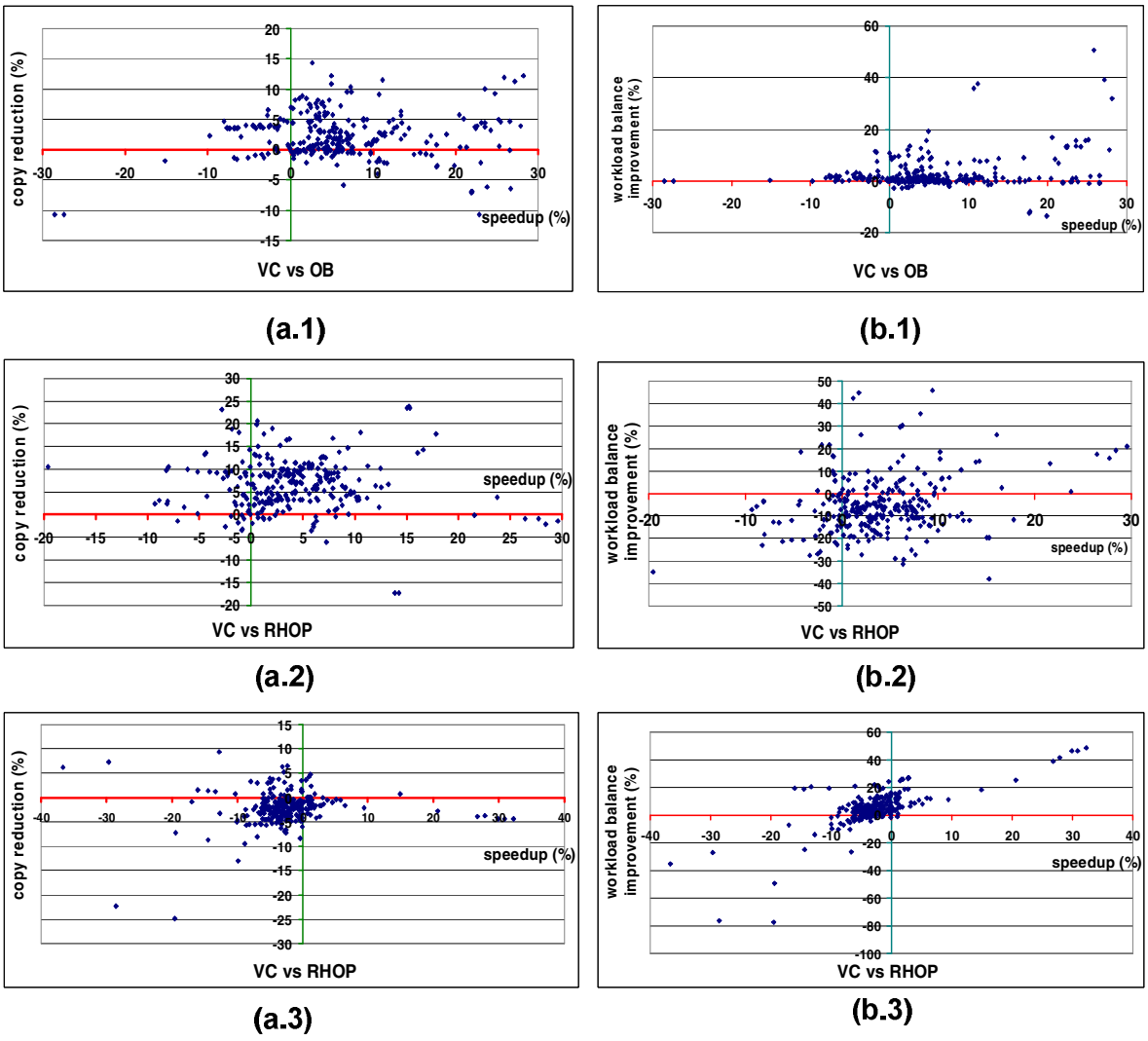

Figure 6. Comparisons of copy reduction and workload balance improvement

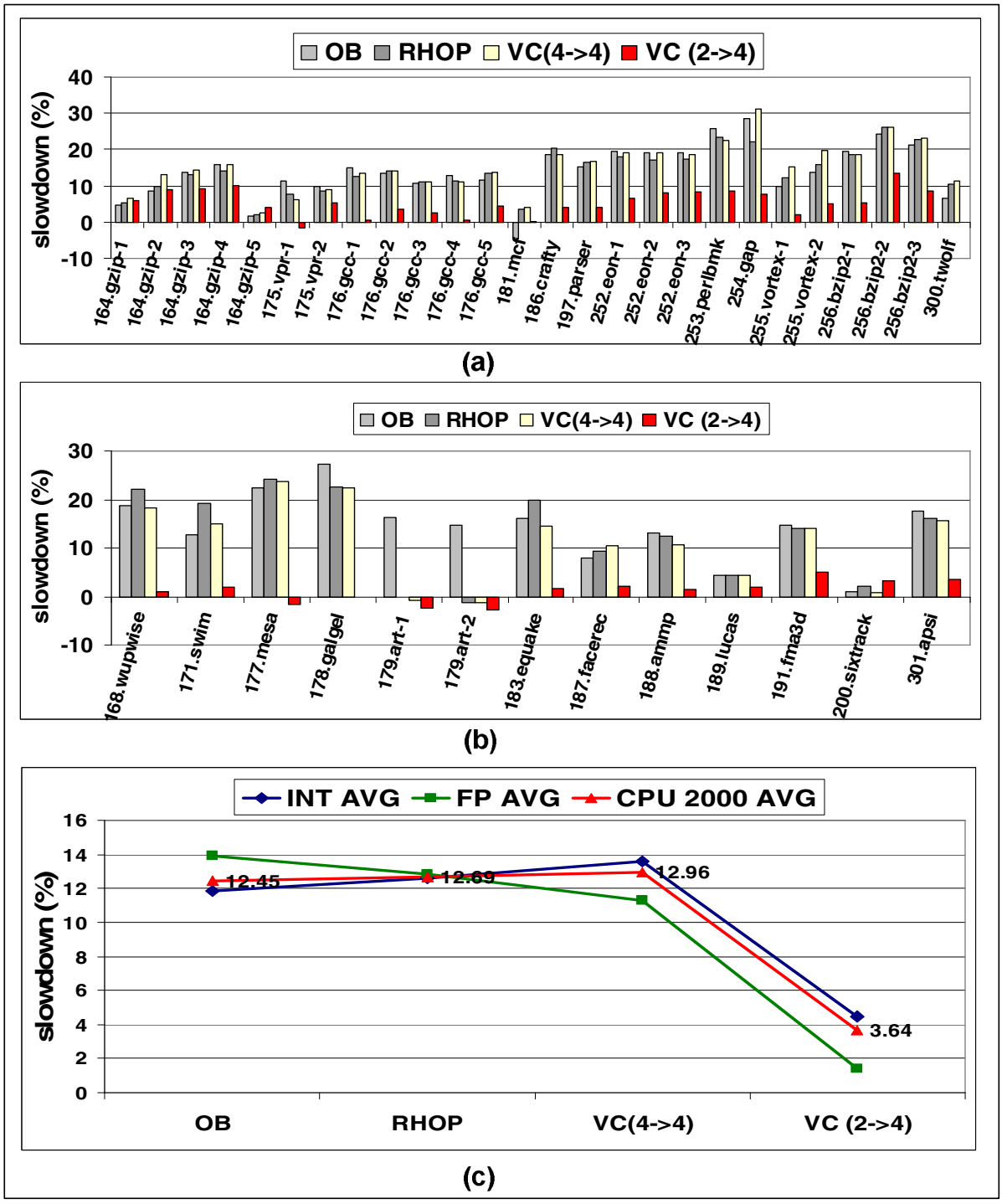

Figure 7. The performance slowdown with respect to the best hardware-only steering on 4-cluster microarchitectures: (a) SpecInt 2000 (b) SpecIntFP 2000 (c) the average results.

draw is that VC obtains better performance than RHOP. This improvement is due to the higher reduction in the number of copy instructions generated by VC. However, VC has worse workload balance than RHOP in most of the cases. Hence this result clearly demonstrates that our hybrid algorithm makes better tradeoff between the number of copies and the workload balance. In particular, VC can send critical dependence chains to one single cluster, which means no penalties due to copies, at the expense on increasing workload imbalance. Moreover, it also implies that copy reduction tends to be more important than workload balance for most of the benchmarks.

The importance of the copy reduction observed in the above comparison between VC and RHOP is also demonstrated when comparing VC and OP. Figures 6(a.3) and (b.3) show the results for this comparison. VC obtains better workload balance but generates more copies than OP for most of the cases. The main reason why OP is performing better than VC is the fact that it tends to give higher priority to communication cost at runtime.

5.4. Scalability

We have already demonstrated that our hybrid approach can achieve almost the same performance as its hardware-only approach for a 2-cluster machine. In this section we present results for a 4-cluster machine in order to demonstrate the scalability of our approach.

In Figure 7 performance results are shown for OB, RHOP and VC configurations compared with OP. Moreover, the results for two configurations of VC algorithm are presented. These two configurations differ in the number of virtual clusters. In particular, the first configuration (VC($4 \rightarrow 4$)) uses 4 virtual clusters, while the other one uses only 2 virtual clusters (VC($2 \rightarrow 4$)).

The main observation we can draw from Figure 7 is that VC($2 \rightarrow 4$) performs significantly better than OB, RHOP and VC($4 \rightarrow 4$). At the same time, VC($2 \rightarrow 4$) only has 3.64% slowdown compared with the best hardware-only steering algorithm.

Furthermore, Figure 7 shows that when moving from VC($4 \rightarrow 4$) to VC($2 \rightarrow 4$), almost 10% speedup is obtained. The main reason why VC($4 \rightarrow 4$) performs worse than VC($2 \rightarrow 4$) is that pairs of critical and dependent instructions that naturally should go to the same physical cluster are spread among several virtual clusters. The hardware side of our hybrid approach may map them to two different physical clusters according to the dynamic workload of each cluster, which generate extra communication penalties. According to our experiments, the VC($4 \rightarrow 4$) configuration generates 28% more copy instructions than VC($2 \rightarrow 4$) on average.

6. Conclusion

The clustered architecture is a promising architecture for next-generation microprocessors, as it circumvents many complexity, power and thermal problems. One of the most important units in the clustered architecture is the steering unit that distributes instructions to the clusters at the back-end. Although there exists several hardware-only steering implementations that show good performance, the hardware complexity involved is high, which makes it difficult to be implemented because of the tight timing constraint of the pipeline. In this paper, we address this complexity issue and propose a novel software-hardware hybrid algorithm.

The proposed algorithm makes the compiler and the hardware work together by using the concept of virtual clustering. The compiler performs the initial steering and assigns a virtual cluster number for each instruction based on the static properties of the programs such as data dependences. At runtime, the hardware checks the dynamic workload information and maps the virtual cluster to the physical cluster. By doing so, we remove most of the hardware complexity and achieve almost the same performance as its hardware-only counterpart.

Finally, we have compared our algorithm against two state-of-the-art software-only algorithms. The experiment shows that the hybrid algorithm obtains almost 5% and 10% speedups on average for 2-cluster and 4-cluster machines, respectively.

7. Acknowledgements

This work has been partially supported by the Spanish Ministry of Education and Science under grants TIN2004-03702 and TIN2007-61763 and Feder Funds. We would like to thank the referees and David Kaeli for their helpful comments and suggestions. We would also like to thank Harish Patil of Intel for answering our PinPoint-related questions.

References

[1] A. Aletà, J. M. Codina, J. Sánchez, and A. González. Graph-partitioning based instruction scheduling for clustered processors. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 150–159, Washington, DC, USA, 2001. IEEE Computer Society.

[2] A. Aletà, J. M. Codina, J. Sánchez, A. González, and D. R. Kaeli. Exploiting pseudo-schedules to guide data dependence graph partitioning. In *PACT '02: Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 281–290, Washington, DC, USA, 2002. IEEE Computer Society.

[3] A. Baniasadi and A. Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 337–347, New York, NY, USA, 2000. ACM.

[4] D. Burger, S. Keckler, K. McKinley, M. Dahlin, L. John, C. Lin, C. Moore, J. Burrill, R. McDonald, and W. Yoder. Scaling to the end of silicon with edge architectures. *Computer*, 37(7):44–55, July 2004.

[5] R. Canal, J. Parcerisa, and A. Gonzalez. Dynamic cluster assignment mechanisms. *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on*, pages 133–142, 2000.

[6] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for vliws: a preliminary analysis of tradeoffs. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 292–300, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[7] P. Chaparro, J. Gonzalez, and A. Gonzalez. Thermal-aware clustered microarchitectures. *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*, pages 48–53, 11-13 Oct. 2004.

[8] M. Chu, K. Fan, and S. Mahlke. Region-based hierarchical operation partitioning for multicluster processors. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 300–311, New York, NY, USA, 2003. ACM.

[9] J. M. Codina, J. Llosa, and A. González. A comparative study of modulo scheduling techniques. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 97–106, New York, NY, USA, 2002. ACM.

[10] J. M. Codina, J. Sanchez, and A. Gonzalez. A unified modulo scheduling and register allocation technique for clustered processors. *pact*, 00:0175, 2001.

[11] J. M. Codina, J. Sanchez, and A. Gonzalez. Virtual cluster scheduling through the scheduling graph. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 89–101, Washington, DC, USA, 2007. IEEE Computer Society.

[12] G. Desoli. Instruction assignment for clustered vliw dsp compilers: A new approach. Technical report, HP Laboratories, 1998.

[13] R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, 1986.

[14] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The multicluster architecture: reducing cycle time through partitioning. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 149–159, Washington, DC, USA, 1997. IEEE Computer Society.

[15] J. González, F. Latorre, and A. González. Cache organizations for clustered microarchitectures. In *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues*, pages 46–55, New York, NY, USA, 2004. ACM.

[16] S. Jang, S. Carr, P. Sweany, and D. Kuras. A code generation framework for vliw architectures with partitioned register banks. In *Pro. of 3rd Int. Conf. on Massively Parallel Computing Systems*, 1998.

[17] K. Kailas, A. Agrawala, and K. Ebcioglu. Cars :a new code generation framework for clustered ilp processors. *hpca*, 00:0133, 2001.

[18] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. *Supercomputing*, 00:29, 1995.

[19] R. Nagarajan, S. K. Kushwaha, D. Burger, K. S. McKinley, C. Lin, and S. W. Keckler. Static placement, dynamic issue (spdi) scheduling for edge architectures. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 74–84, Washington, DC, USA, 2004. IEEE Computer Society.

[20] E. Nystrom and A. E. Eichenberger. Effective cluster assignment for modulo scheduling. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 103–114, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[21] E. Özer, S. Banerjia, and T. M. Conte. Unified assign and schedule: a new approach to scheduling for clustered register file microarchitectures. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 308–315, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[22] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 81–92, Washington, DC, USA, 2004. IEEE Computer Society.

[23] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *SIGMICRO Newsl.*, 12(4):183–198, 1981.

[24] P. Salverda and C. Zilles. A criticality analysis of clustering in superscalar processors. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 55–66, Washington, DC, USA, 2005. IEEE Computer Society.

[25] J. Sánchez and A. González. The effectiveness of loop unrolling for modulo scheduling in clustered vliw architectures. In *ICPP '00: Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*, page 555, Washington, DC, USA, 2000. IEEE Computer Society.

[26] S. S. Sastry, S. Palacharla, and J. E. Smith. Exploiting idle floating-point resources for integer execution. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 118–129, New York, NY, USA, 1998. ACM.