Cluster Prefetch: Tolerating On-Chip Wire Delays in Clustered Microarchitectures

Rajeev Balasubramonian
School of Computing, University of Utah

ABSTRACT

The growing dominance of wire delays at future technology points renders a microprocessor communication-bound. Clustered microarchitectures allow most dependence chains to execute without being affected by long on-chip wire latencies. They also allow faster clock speeds and reduce design complexity, thereby emerging as a popular design choice for future microprocessors. However, a centralized data cache threatens to be the primary bottleneck in highly clustered systems. The paper attempts to identify the most complexity-effective approach to alleviate this bottleneck. While decentralized cache organizations have been proposed, they introduce excessive logic and wiring complexity. The paper evaluates if the performance gains of a decentralized cache are worth the increase in complexity. We also introduce and evaluate the behavior of *Cluster Prefetch* - the forwarding of data values to a cluster through accurate address prediction. Our results show that the success of this technique depends on accurate speculation across unresolved stores. The technique applies for a wide class of processor models and most importantly, it allows high performance even while employing a simple centralized data cache. We conclude that address prediction holds more promise for future wire-delay-limited processors than decentralized cache organizations.

Categories and Subject Descriptors

C.1.1 [**Processor Architecture**]: Single Data Stream Architectures; B.3.m [**Memory Structure**]: Miscellaneous

General Terms

Processor, Design

Keywords

Clustered microarchitectures, distributed caches, communication-bound processors, effective address and memory dependence prediction, data prefetch

1. INTRODUCTION

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'04, June 26–July 1, 2004, Saint-Malo, France.
Copyright 2004 ACM 1-58113-839-3/04/0006 ...$5.00.

Improvements in process technology have resulted in dramatic increases in clock speeds and transistor budgets. However, delays across wires have not been scaling down at the same rate as delays across transistor logic. Projections have shown that in 10 years, the delay in sending a signal across the diameter of a chip can be of the order of 30 cycles [1]. The consequence of these technology trends is that microprocessors in the billion-transistor era are likely to be increasingly *communication-bound*.

In an effort to deal with long wire latencies, architects have proposed clustered microarchitectures [11, 14, 20, 27, 31, 38]. A clustered design helps reduce the size, porting, and bandwidth requirements of many processor structures, thereby enabling faster clocks and reducing design complexity and power consumption. Variants of these designs have even been incorporated in real microprocessors such as the Alpha 21264 [21] and the Pentium 4 [18].

In a clustered microarchitecture, each cluster has a limited set of functional units, registers, and issue queue entries. Instruction fetch, decode, and rename are centralized, after which instructions are distributed among the clusters. Operand bypass within a cluster happens in a single cycle. Hence, as far as possible, dependent instructions are steered to the same cluster. However, instructions in a cluster frequently read operands produced in a different cluster. This communication of operands between clusters can take up a number of cycles and can limit instruction per cycle throughput. In spite of this, a clustered microarchitecture is an extremely attractive option because of its potential for a faster clock, lower power, and lower design effort.

While a clustered design helps localize communication to a limited set of functional units, the evaluation of the effect of technology trends, such as increased transistor budgets, longer wire latencies, and the shift to multi-threaded workloads, on its behavior has received little attention. The most prominent effect of these trends is an increase in the total number of on-chip clusters. This is motivated not only by the need to improve single-thread performance, but also by the recent emphasis on the extraction of thread-level parallelism (Intel's Pentium 4 [18] and IBM's Power4 [36]). This is facilitated by large transistor budgets and the reuseability of a cluster design, which allows the implementation and verification complexity to not scale in proportion to the increased area. A second related effect is that instructions of a program might spend tens of cycles communicating their input and output values across widely separated structures on the chip. This is especially true for load or store instructions, which require transfers of the address *and* data between a cluster and the cache.

Most recent studies on clustered designs have focused on instruction distribution algorithms for up to four clusters with modest inter-cluster latencies, a centralized data cache, and zero commu-

nication cost between the cache and clusters. Current technology trends are likely to quickly render these assumptions invalid. In the near future, a cluster would likely have to spend many cycles sending an address to a centralized data cache, and an equal amount of time to receive the data back. This phenomenon is likely to represent one of the biggest bottlenecks for CPU performance.

The focus of this paper is the study of design considerations for the L1 data cache in a highly clustered and wire-delay-limited microprocessor. Our hypothesis is that a decentralized cache organization entails too much design complexity and yields little benefit. We claim that prediction mechanisms have the potential to hide long wire delays and can result in simpler designs. Thus, the paper directly contrasts two different approaches to designing a high performance cache organization and provides data that might help shape future research directions.

We start by using a centralized L1 data cache as our base case. While such a design choice reduces the implementation complexity, the long access latencies experienced by distant clusters limit its performance. We analyze the average lifetime of a load and observe that a large fraction of time is spent transferring addresses and data between the cache and the clusters, and waiting for store addresses to arrive at the cache. To alleviate this bottleneck, we propose *Cluster Prefetch* – the transfer of data from the cache to a cluster before the cluster initiates a request. Prefetching has been commonly employed in higher levels of caches to hide long access latencies. With the emergence of long communication latencies within the CPU itself, it is necessary to employ prefetching techniques to bring data in from the L1. We observe that load address prediction can be quite accurate for many programs with high instruction level parallelism (ILP), and the relevant data can be forwarded to a cluster soon after the instruction is decoded. However, to not violate memory dependences, the prefetch can issue only when earlier store addresses have been resolved. Note that prefetch from L2 or beyond is typically free from such constraints. Hence, the success of *cluster prefetch* depends strongly on being able to correctly speculate ahead of unresolved stores. By predicting store addresses and memory dependences, we are able to quickly initiate prefetch requests and hide communication latencies. Our results demonstrate that *cluster prefetch* – the synergistic combination of memory dependence speculation and load address prediction and prefetch – significantly improves performance for a number of high-ILP programs.

Next, we evaluate the potential of a distributed cache – an alternative approach to reducing communication latencies. While this increases implementation complexity, a load can now be steered close to its data, thereby reducing its average communication cost. However, to maintain correctness, store addresses have to be broadcast to the entire processor. Thus, the biggest bottleneck in the system continues to be the latency involved in resolving memory dependences. As a result, a decentralized cache organization fails to outperform a centralized cache combined with *cluster prefetch*.

We verify that *cluster prefetch* is an effective approach for a wide variety of processor parameters. It entails very little overhead in terms of transistors or design complexity and yields performance improvements of more than 8% for half the programs studied.

The rest of the paper is organized as follows. In Sections 2 and 3, we describe our base clustered processor and our simulation infrastructure. Section 4 analyzes the behavior of memory speculation and load address prediction for a system with a centralized L1 data cache. In Section 5, we evaluate the potential of decentralized data caches. We outline related work in Section 6 and conclude in Section 7.

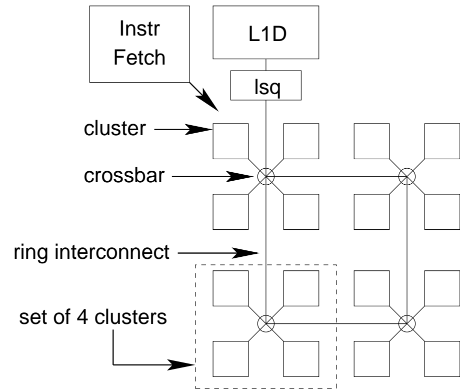

Figure 1: The 16-cluster system with four sets of four clusters each. A crossbar interconnect is used for communication within a set of clusters and a ring connects the four crossbar routers.

2. THE BASE CLUSTERED PROCESSOR

A number of architectures have been proposed to exploit large transistor budgets on a single chip [7, 19, 25, 26, 27, 33]. Most billion-transistor architectures partition the processor into multiple small computational units and distribute instructions of a single application across these units (either statically or dynamically). The high cost of wire delays in future technologies renders many of these architectures *communication-bound*. In this paper, as an evaluation framework, we employ a dynamically scheduled clustered processor. The results from this study can apply to other partitioned architectures as well.

Partitioned Execute Engine. Our clustered organization resembles those proposed in prior bodies of work [2, 5, 6, 11, 38]. In such a processor (Figure 1), branch prediction, instruction fetch, and register renaming are centralized operations. During register renaming, the instruction is assigned to one of many clusters. Each cluster has a small issue queue, physical register file, and a limited number of functional units with a single cycle bypass network among them. If an instruction's source operands are in a different cluster, copy instructions are inserted in the producing clusters and the values are copied into physical registers in the consuming cluster. This copy can take up a number of cycles.

Centralized Cache. For a load or store, the effective address is computed in one of the clusters and then sent to a centralized load/store queue (LSQ). The LSQ checks for memory dependences before issuing loads or stores. A load accesses the cache only when it is known to not conflict with earlier stores, *i.e.*, all earlier store addresses are known. The data read from the cache is then forwarded back to the cluster that issued the load. A store writes data to the cache when it commits. Thus, each load involves one address transfer to the centralized LSQ and one data transfer back to the cluster. Each store involves one address and one data transfer to the LSQ. In our layout, we assume that the centralized cache and centralized instruction decode are located close to each other.

Our clustered organization incorporates many recent innovations and represents the state-of-the-art, as outlined below.

Instruction Steering Heuristic. The instruction per cycle (IPC) throughput of a clustered processor is less than that of a monolithic processor with no wire delay penalties and with the same number of resources. This is because of communication cycles between instructions and because of higher contention for functional units in a clustered processor. Hence, the primary goal of the instruction

steering heuristic is to minimize communication stalls and maximize load balance. Our steering heuristic is based on recent proposals [11] and can be implemented with a modest amount of logic. For each instruction, the heuristic computes a "suitability metric" for each cluster and assigns the instruction to the cluster with the highest "suitability". For example, a cluster that produces a source operand for that instruction, is considered "more suitable". Similarly, a cluster that already has many instructions assigned to it, is considered "less suitable". Based on these factors, for each instruction, weights are assigned to each cluster and the heuristic assigns the instruction to the cluster with the highest weight. We also use a criticality predictor [37] to detect the source operand that is produced later and assign a higher weight to that producing cluster. For loads, clusters that are close to the centralized cache are assigned a higher weight. If the selected cluster is full, the instruction is steered to the nearest cluster. A number of experiments were conducted to determine the values of the weights for each simulated model.

Interconnect. Aggarwal and Franklin [3] point out that a crossbar has better performance when connecting a small number of clusters, while a ring interconnect performs better when the number of clusters is increased. To take advantage of both characteristics, they propose a hierarchical interconnect, where a crossbar connects four clusters and a ring connects multiple sets of four clusters. This allows low-latency communication between nearby clusters. Our 16-cluster processor model, shown in Figure 1, has four sets of clusters, each set consisting of four clusters.

Note that each edge in the hierarchical interconnect consists of two 64-wide unidirectional links that are used for communication of register values, load/store addresses, and load/store data. This is referred to as the *Primary Interconnect*. Since a large number of accesses are made to the data caches, the links leading in and out of them have twice the bandwidth of other links. A separate interconnect is used for transferring decoded instructions to the respective clusters, referred to as the *Instruction Delivery Interconnect*. A much narrower interconnect is used to transfer other control signals, such as branch mispredicts, instruction completion, etc. Latencies on all of these interconnects are modeled in our simulator.

Parameter	Value
Fetch queue size	64
Branch predictor	comb. of bimodal and 2-level
Bimodal predictor size	2048
Level 1 predictor	1024 entries, history 10
Level 2 predictor	4096 entries
BTB size	2048 sets, 2-way
Branch mpred penalty	at least 12 cycles
Fetch width	8 (across up to two basic blocks)
Dispatch/commit width	16
Issue queue size	15 in each cluster (int and fp, each)
Register file size	30 in each cluster (int and fp, each)
ROB size	480
Integer ALUs/mult-div	1/1 (in each cluster)
FP ALUs/mult-div	1/1 (in each cluster)
L1 I-cache	32KB 2-way
L1 D-cache	32KB 2-way set-associative, 6 cycles, 4-way word-interleaved
L2 unified cache	2MB 8-way, 25 cycles
TLB	128 entries, 8KB page size (I and D)
Memory latency	160 cycles for the first chunk

Table 1: Simplescalar simulator parameters.

Benchmark	Base IPC	L1 data cache miss rate
applu	2.21	9.9
apsi	2.55	9.9
art	1.54	25.5
equake	4.13	0.0
fma3d	2.45	0.1
galgel	3.60	0.2
lucas	2.03	17.7
mesa	3.27	1.1
mgrid	2.39	4.5
swim	1.85	29.6
wupwise	2.39	1.5

Table 2: Benchmark description. Baseline IPC is for a monolithic processor with as many resources as the 16-cluster system and no wire delay penalties. The L1 data cache miss rate is for a 32KB 2-way cache.

3. METHODOLOGY

3.1 Simulation Parameters

Our simulator is based on Simplescalar-3.0 [10] for the Alpha AXP instruction set. Separate issue queues and physical register files are modeled for each cluster. Contention on the interconnects and for memory hierarchy resources (ports, banks, buffers, etc.) are modeled in detail.

To model a wire-delay-constrained processor, each of the 16 clusters is assumed to have 30 physical registers (int and fp, each), 15 issue queue entries (int and fp, each), and one functional unit of each kind. In a later section, we verify that our results are not very sensitive to this choice of parameters. While we do not model a trace cache, we fetch instructions from up to two basic blocks in a cycle. We use the methodology used by Agarwal *et al.* [1] to estimate clock speeds and memory latencies, following SIA roadmap projections. We use CACTI-3.0 [35] to estimate access times for cache organizations. Many different organizations were simulated for a diverse benchmark set before determining the best base case. The L1 cache has four word-interleaved banks with a total capacity of 32KB. Important simulation parameters are listed in Table 1.

The latencies on the interconnect would depend greatly on the technology and processor layout. We assumed the following latencies: a cycle to send data to the crossbar router, a cycle to receive data from the crossbar router, and four cycles to send data between crossbar routers. Thus, the two most distant clusters on the chip are reachable in 10 cycles. Considering that Agarwal *et al.* [1] project 30-cycle worst-case on-chip latencies at 0.035μ technology, we expect this choice of latencies to be representative of wire-limited future microprocessors[1]. To further validate our proposals, we also present results for other latency choices. We assume that each communication link is fully pipelined, allowing the initiation of a new transfer every cycle. We also assume unlimited buffer space at each of the routers. A recent study by Parcerisa *et al.* [29] shows that such an assumption is not unreasonable and that these buffers typically require only eight entries.

3.2 Benchmark Set

Billion transistor architectures are best suited to workloads that yield high parallelism or are multi-threaded. In this paper, our analysis focuses on high parallelism programs, such as those found in the SPEC-FP benchmark suite. While we do not evaluate multi-threaded workloads, as part of our sensitivity analysis, we present data on SPEC-Int programs to evaluate the effect of our proposals on code that is not as regular as floating-point code.

For most experiments, we use 11 of the 14 SPEC2k floating-

[1] It must be noted that the L2 would account for a large fraction of chip area.

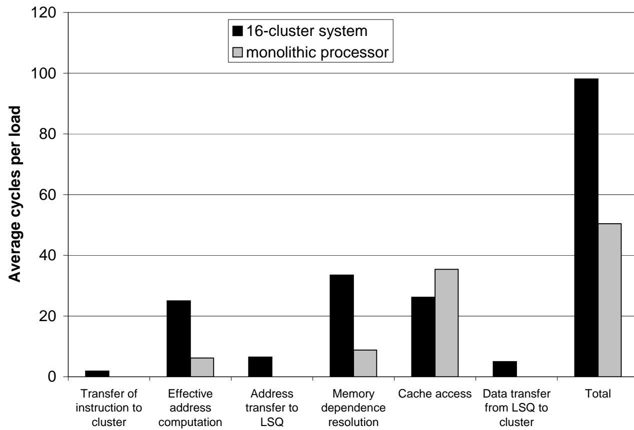

Figure 2: Average number of cycles spent by a load in different stages of the pipeline for a 16-cluster system and for a monolithic processor with the same resources and no communication penalties.

point programs [2]. These programs were fast-forwarded for two billion instructions, simulated in detail for a million instructions to warm up various processor structures, and then measured over the subsequent 100 million instructions. The reference input set was used for all programs. Table 2 lists the programs, their IPCs for a monolithic processor with no wire delay penalties and as many resources as the base case, and their L1 data cache miss rates.

4. CLUSTER PREFETCH

One of the primary goals of the paper is to evaluate if prediction techniques can help hide long communication latencies in a highly clustered processor. The success of such an approach can eliminate the need for decentralized cache organizations, that inevitably lead to complex and power-inefficient designs.

4.1 Lifetime of a Load

We begin by examining where an average load spends its time on the 16-cluster system described in Section 3. Figure 2 graphs the average number of cycles spent in each logical phase: instruction transfer to the cluster after decode, effective address computation, transfer of the effective address to the centralized LSQ, resolution of memory dependences, cache access, and transfer of data back to the cluster, respectively. For the clustered processor (the black bars), an average load consumes 98.1 cycles between decode and instruction completion. We observe that resolution of memory dependences and transfer of effective address and data take up 45 cycles, nearly half the total execution time for an average load. If the centralized LSQ could predict the effective address at instruction decode time, instantly resolve the memory dependences, execute the cache access, and transfer the data, it would arrive at the cluster after 31.2 cycles. Thus, a load, on average, could complete nearly 67 cycles earlier.

To contrast this behavior with that seen in traditional systems, the grey bars in Figure 2 show the corresponding latencies in a conventional monolithic processor with the same total resources and no communication penalties. An average load completes in half the time it takes on a clustered processor. We observe that the time taken to compute a load's effective address and resolve its memory dependences goes down by a factor of four. This is because the dependence chains leading up to the load and store address computations encounter zero communication costs and execute quickly. Further, there is no communication cost in transferring the store address to the LSQ. The cache access time goes up slightly. This is caused by multiple loads waiting for the same block to be fetched from memory. In a clustered processor, since the rate of instruction issue is much slower, prior loads prefetch a block for the later loads and they experience shorter cache access latencies. In the monolithic processor, the instruction consumes 50.4 cycles between decode and completion. If we were to predict the effective address at instruction decode time, instantly resolve the memory dependences, and execute the cache access, data would be available after 35.4 cycles. Thus, these techniques would save only 15 cycles for each load on a conventional processor, but save 67 cycles on a clustered processor.

4.2 Address Prediction and Memory Dependence Speculation

The results from the previous subsection clearly highlight the emergence of new bottlenecks in future microprocessors. Effective address computation, address and data communication, and memory dependence resolution, each take up about the same time as the cache access itself. To reduce design complexity, we preserve the centralized nature of the cache organization and explore the potential of prediction techniques in alleviating the communication bottlenecks.

Address Prediction and Prefetch. At the decode stage, we access a simple address predictor to predict the effective address of a load. Loads that do not have a high-confidence prediction go through the same pipeline as before. If a high-confidence prediction is made, the predicted address is placed in the centralized LSQ where it awaits the resolution of memory dependences. When memory dependences are resolved in the centralized LSQ, the load issues and the fetched data is forwarded to the instruction's cluster. Once the instruction computes its effective address, it awaits the arrival of its data. If data has been already prefetched, the load completes in a single cycle and awakens its dependents. The computed effective address is sent to the centralized LSQ to verify that the prediction was correct. If the prediction was incorrect, the correct data has to be forwarded and all dependent instructions have to be squashed and re-executed. To reduce the implementation complexity, we adopt a mechanism very similar to a branch mispredict and squash all instructions after the address mispredict. By restricting the prefetch to high-confidence predictions, the number of mispredicts is kept to a minimum.

Since data can be received by a cluster even before the effective address is computed, it is possible to initiate the dependent instructions early. However, we make a conscious decision not to implement this feature. Address prediction proposals in the past [4, 8, 9, 17, 32, 34] have exploited this feature to hide the latency of executing long dependence chains. Unlike earlier studies, our focus here is the effect of these techniques on hiding communication delays. To isolate only the performance improvements from hiding these long communication latencies, we await the computation of the effective address before waking up dependent instructions. Since dependent instructions are woken up before the address prediction gets verified at the LSQ, squash and recovery is required on an address mispredict.

Address Predictor. To reduce design complexity, we employ a simple strided address predictor [12]. The load PC indexes into a table that keeps track of the stride and the last predicted address. Every branch mispredict clears the "last predicted address" field

[2] *Sixtrack* and *Facerec* were not compatible with our simulator and *Ammp* is extremely memory bound, has an IPC of less than 0.1, and is unaffected by CPU optimizations.

and requires that we track the next few accesses to determine its value again. To initially compute the stride, we examine five consecutive accesses to verify a consistent stride. If more than five incorrect predictions are made for a load, its fields are cleared and the stride is re-computed. At the time of decode, if the stride and last predicted address are unknown, no prediction is made. Such an implementation ensures that only high-confidence predictions are made. Note that every address mispredict behaves like a branch mispredict and must be kept to a minimum.

It is possible to build complex address predictors that might yield higher prediction accuracies. However, our objective is to evaluate if a centralized cache augmented with *simple* prediction techniques can work as well as a more complex decentralized cache organization. We explicitly choose a simple predictor implementation in an effort to fulfil our goals of low complexity *and* high performance. The design of low complexity and high accuracy address predictors and their impact on communication-bound processors is interesting future work.

Steering Heuristic. We modify the instruction steering heuristic to further reduce communication costs. Clusters close to the centralized LSQ continue to receive a higher weight for loads that do not have their address predicted. However, this is not done for loads that are able to predict their effective address – prefetching their data allows these loads to tolerate longer communication latencies.

Memory Dependence Speculation. The ability to accurately predict load addresses is meaningless if loads spend most of their time in the centralized LSQ awaiting the resolution of earlier store addresses. The communication costs in a clustered processor prolong the store address computation and communication time, resulting in longer waits in the LSQ for all loads. To mitigate this problem, we implement the following two simple prediction techniques.

The first technique uses the load address predictor to also predict addresses for stores. These predicted addresses are used to quickly resolve memory dependences. If there is an address mispredict, all instructions following the store are squashed and re-fetched, similar to branch mispredict recovery.

The second technique identifies if a store is likely to pose conflicts with subsequent loads [22]. Every time a store in the LSQ forwards its data to a later load, the store PC is used to index into a table and set a bit, indicating its likelihood to pose conflicts in the future. If a store does not have its corresponding bit set in the table, we allow loads to issue even before the store address is resolved. If there is a mispredict, all instructions following the store are squashed and re-fetched. Note that the two techniques are orthogonal – a store address may be hard to predict, but its likelihood to conflict with later loads may be easily predicted. As before, we consciously choose simple implementations instead of a more complex and potentially more accurate predictor.

These two techniques allow us to exploit load address prediction and prefetch data into a cluster in time for its use. Our results show that each of these techniques by itself does not result in significant performance improvements, but they interact synergistically with each other. The combination of load address prediction, store address prediction, store-load conflict prediction, and the early forwarding of data is referred to as *Cluster Prefetch*.

4.3 Results

In Figure 3, we demonstrate the performance improvements from adding these techniques. The first bar in the figure represents instructions per cycle (IPC) performance for the base case with 16 clusters and the centralized data cache. The second bar uses the

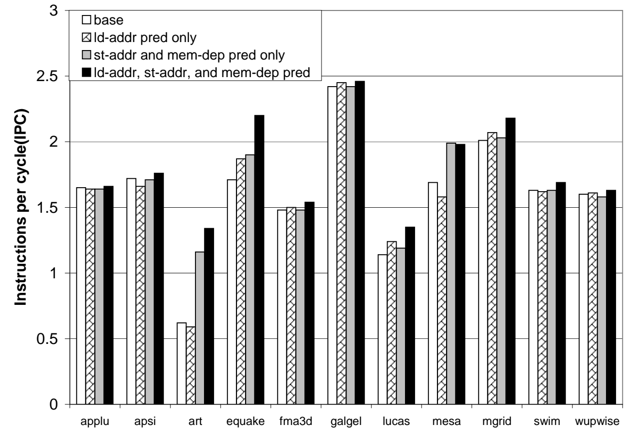

Figure 3: IPCs for the base clustered processor with the centralized cache with various prediction techniques. The second bar includes the use of load address prediction only, the third bar only includes store address prediction and memory dependence prediction, while the last bar includes all three techniques.

load address predictor to forward data early to a cluster and hide the cost of communication between the cluster and the centralized LSQ. The third bar does not use the load address predictor, but uses the two memory dependence speculation techniques – store address prediction and identifying stores that are likely to not pose conflicts. The last bar combines all three techniques. Table 3 provides various statistics that help us better understand the performance changes.

From the figure, we see that in many cases, the performance improvement from load address prediction alone is marginal. Table 3 reveals that accurate load address prediction can be achieved for many programs, but only *equake* and *lucas* are able to exploit it and register notable improvements. *Lucas* is able to successfully predict nearly 100% of its loads, and *equake* is able to do the same for 76% of its loads. Often, a load cannot issue early because it is waiting for a prior store address. This is a frequent occurrence in *art* and *mgrid*, that have high load address prediction accuracies (greater than 95%), but no corresponding performance increase. In fact, a number of programs (including *art*) display slightly poorer performance. This happens because of the bursty nature of instruction fetch, load address prediction, and data forwarding. The contention that this creates on the interconnect slows down many dependence chains. The most noticeable slowdown is observed for *mesa*. In *mesa*, more than 4M additional transfers between the cache and cluster are effected than in the base case because of address prediction along branch mispredicted paths. Further, the 4K load address mispredicts, and resulting recovery, delay instruction fetch by a total of about 0.9M cycles. The cost of address mispredict recovery has a noticeable effect in *apsi* as well.

By comparing the first and third bars, we examine the effect of memory speculation techniques alone. These help alleviate the bottleneck created by long dependence chains and communications involved in the computation of store addresses. Again, in spite of high prediction accuracies for most programs, *art*, *equake*, and *mesa* are the only programs to show noticeable performance improvements – these are the programs where loads spend the longest time awaiting the resolution of store addresses. Again, mispredicts are few in number and result in negligible recovery overhead.

Finally, by combining load address prediction and memory speculation, we observe a more than additive increase in performance

Benchmark	Num loads	Num predicted	Mispredicts	Num stores	Num predicted	Mispredicts	Predicted store-load non-conflict	Mispredicts	Overall IPC improvement
applu	22M	7.8M	0	8M	3.7M	0	2.7M	80	1%
apsi	27M	20M	71K	12M	8.4M	36K	9M	288	2%
art	33M	31.2M	13K	4M	4M	3K	1M	58	116%
equake	27M	20.5M	163	7M	7M	23	1.6M	113	29%
fma3d	15M	3.8M	7K	2M	1.8M	6	1.2M	173	4%
galgel	35M	1.7M	0	15K	0	0	0	0	2%
lucas	15M	15M	679	8M	8M	0	7.9M	53	18%
mesa	26M	16.5M	4K	10M	9.5M	511	1.7M	314	17%
mgrid	38M	37.3M	124	1.5M	1.4M	0	1.5M	61	8%
swim	29M	28.7M	143	4.2M	4.2M	2	4.2M	83	4%
wupwise	20M	4.7M	150	7M	2.2M	0	7M	91	2%

Table 3: Statistics demonstrating the effectiveness of load and store address prediction and store-load conflict prediction. Overall IPC improvement is that seen when employing all three techniques together.

improvements. Memory speculation allows more address predictions to issue early. For example, in *equake*, the IPC improvement from address prediction is 9%, from memory speculation is 11%, and from the combination is 29%. Significant improvements are also observed for *art* (116%), *lucas* (18%), *mesa* (17%), and *mgrid* (8%). When comparing the harmonic means (HM) of IPCs, the overall improvement over the base case is 21%. We were unable to improve performance in some of the programs because of poor load address prediction rates (*applu*, *fma3d*, *galgel*, and *wupwise*). While *swim* was highly predictable, its performance is limited by L2 and memory latencies, not by L1 access latencies. In fact, its performance is already very close to that of the monolithic processor, and our optimizations have very little impact.

When these same techniques are implemented on a conventional monolithic processor with no wire-delay penalties, negligible improvements are observed. Recall that we do not forward prefetched data to the load's dependents until the effective address is computed. The overall improvement from load address prediction and memory speculation was less than 3%, with a maximum improvement of 16% seen for *equake*. Thus, simple implementations of these techniques that yield little improvement in conventional microprocessors, have a significant impact in highly clustered and communication-bound processors of the future.

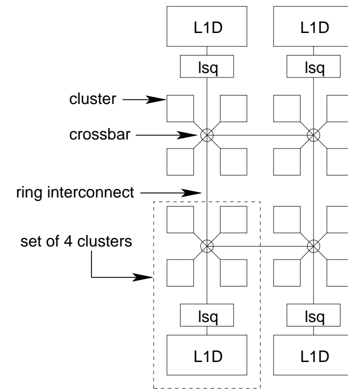

Figure 4: The 16-cluster system with four sets of four clusters each and a distributed LSQ and L1 data cache. Each set of four clusters is associated with an LSQ and cache bank.

5. DECENTRALIZED DATA CACHES

The previous section discusses how address prediction and memory speculation allow us to prefetch data in an effort to hide long on-chip communication latencies. An alternative technique to reducing these delays is to have multiple caches on the die, so that every cluster is close to some cached data. In this section, we evaluate the performance impact of such a distributed cache organization and qualitatively discuss the complexity involved.

5.1 Replicated Cache Banks

The first decentralized organization we examine implements replicated cache banks and LSQs (Figure 4). Every load gets serviced by the cache bank that is closest to it, without ever incurring a communication on the ring interconnect. Such a mechanism also reduces register communication as instruction steering no longer factors in the cache location in its decision-making. The only problem with this approach is the redundancy in write traffic. Every cache block that is fetched from L2 is broadcast to four cache banks. Similarly, every store issued by every cluster has to be broadcast and written to all four cache banks. Including this traffic on the *Primary Interconnect* has a debilitating effect on performance. In our evaluations, we use the *Primary Interconnect* only to tranfer store addresses and data from producing clusters to the nearest LSQ. The broadcast of these addresses and data to the other LSQs happens through a separate *Broadcast Interconnect*. Each cache bank requires as many write ports as the centralized cache bank in the base case.

In terms of hardware and design complexity, the replicated cache organization clearly entails higher overhead than that imposed by address and dependence predictors. The need for an additional *Broadcast Interconnect* increases wiring and layout complexity. Further, datapaths are required between the L2 and every L1 cache bank. There are also many implementation subtleties – ordering of cache replacements to ensure that the cache banks contain identical data, distributed LRU information, etc. Such a design is also likely to have poor power characteristics because of broadcasts for all store addresses and data and multiple writes to each cache bank. What we evaluate here is if the performance improvement merits this increase in complexity.

Figure 5 shows the IPCs for the replicated cache organization. As reference points, the first two bars in the figure present IPCs for the base centralized cache without and with *cluster prefetch*. The third bar represents a replicated cache organization, where each set of four clusters is associated with a 32KB 6-cycle cache bank. The fourth bar includes performance improvement from *cluster prefetch* for the replicated cache organization. The results show that the

reduced cost of communication for a load and the reduced register communication improve performance over the base centralized cache. The overall improvement (while comparing harmonic mean of IPCs) is about 6%, with appreciable speedups being seen for *art* (13%), *equake* (29%), and *mesa* (11%). However, the biggest bottleneck continues to be the time taken to resolve memory dependences. The computation of the store address potentially involves register communications and the computed address has to be broadcast to all the LSQs. As a result, the base replicated cache organization does not perform as well as *cluster prefetch* on a centralized cache organization. When *cluster prefetch* is implemented with the replicated cache, significant speedups are observed. However, the overall improvement over the centralized cache with *cluster prefetch* is marginal (3%). These results indicate that predicting addresses and memory dependences is a more effective technique to handling long communication latencies than moving copies of the data cache closer to the computations.

5.2 Word-Interleaved Distributed Cache

The replicated cache wastes space and incurs multiple writes for every update to data. We study an alternative cache organization that eliminates these problems. Similar to the previous subsection, we employ four cache banks, but each of these banks contains mutually exclusive data. All word addresses of the form 4N are mapped to the first bank, word addresses of the form 4N+1 are mapped to the next, and so on. We assume that each word is eight bytes long. In this organization, when a load address is computed, it is sent to the unique LSQ and cache bank that contains that address. Similarly, store addresses and data are also sent to their respective LSQs. However, to avoid memory conflicts, until the store address is resolved, all the LSQs maintain a dummy entry for every unresolved store. The computation of the store address triggers a broadcast to all the LSQs so they can remove the dummy slot. This broadcast requires a much narrower interconnect than the one used for the replicated cache. Also, store data need not be broadcast. Such an implementation has been used in prior studies [5, 38]. This organization can reduce load communication latencies if the load can be steered close to its data. Since the load address is not known at dispatch time, we employ our address predictor to guide the instruction steering heuristic. If a high-confidence prediction can not be made, the steering heuristic attempts to minimize register communication. However, note that such an organization will continue to incur stall cycles while waiting for store addresses to be resolved. It incurs less implementation complexity and lower power consumption than the replicated cache, but is likely to perform worse because of longer communication latencies for loads. It can yield higher performance only because of its increased cache capacity.

The fifth bar in Figure 5 represents performance for such a decentralized cache organization with a total 128KB L1 capacity. The sixth bar incorporates *cluster prefetch*. We see that in almost all cases, because of imperfect load address prediction, the word-interleaved cache organization performs slightly worse than the replicated cache. Noticeable improvements are seen only in *mesa*, where the larger cache drops the cache miss percentage from 1.1 to 0.2.

5.3 Discussion

The previous sections characterize the behavior of different approaches in handling communication latencies during cache access. A distributed cache organization helps bring the cache closer to the load instruction, but does not target one of the most important bottlenecks in the system – the resolution of memory dependences.

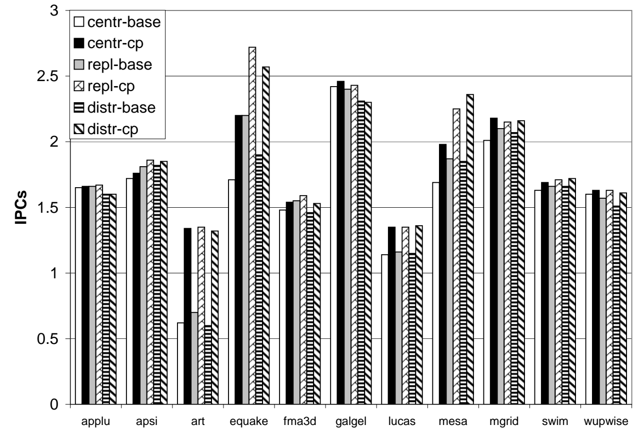

Figure 5: IPCs without and with *cluster prefetch* for three different cache organizations. The first uses a centralized cache, the second uses four replicated cache banks, and the third uses four word-interleaved distributed cache banks.

Memory dependence resolution is slowed by the long latency involved in computing the store effective address and communicating it to the LSQs. Since all the LSQs have to receive this store address, decentralization does not alleviate this bottleneck. Instead, it introduces non-trivial implementation complexity and an increase in power consumption.

We observed that load address prediction and prefetch, and memory speculation provide impressive speedups in all the organizations studied. In fact, with these techniques implemented, cache access ceases to be a major bottleneck and there is very little performance difference between a centralized and decentralized cache. This result has very favorable implications for implementation complexity as a centralized cache is significantly easier to design and consumes less power.

All the designs evaluated in this paper rely on dynamic memory disambiguation for good performance. If the compiler can reliably resolve memory dependences, more efficient architectures can be constructed. For example, instructions and data can be statically mapped to different clusters and cache banks, respectively, and explicit communication instructions can be introduced when memory dependences are detected. Such an organization is likely to have the best performance and modest hardware complexity (interconnects between L2 and multiple L1 cache banks). In some sense, the replicated cache organization with *cluster prefetch* and four times as much cache capacity most closely resembles this static approach in terms of performance – loads do not travel far, do not wait for addresses of unrelated stores, and have access to 128KB of total cache. Our results show that such a replicated cache outperforms a centralized cache with *cluster prefetch* by less than 4%. Thus, the static approach to cache decentralization is likely to not offer a huge performance advantage, when compared with a centralized cache with *cluster prefetch*. Further, it must be noted that only a subset of programs can reliably resolve memory dependences at compile-time.

Address prediction and memory dependence speculation entail negligible overhead. The address predictor is a structure very similar to a branch predictor. Memory dependence speculation requires another table with single-bit entries. The next subsection demonstrates that the total size of these structures can be as little as 18KB, and yet afford good performance. An additional bit is required in

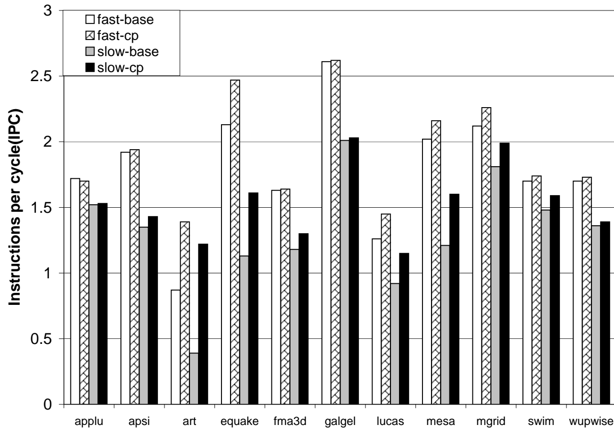

Figure 6: IPCs without and with *cluster prefetch* for clustered processors with Fast and Slow interconnects.

the LSQ to indicate predicted addresses. Each cluster might need a small buffer to store prefetched data. Alternatively, the prefetched data can be directly written to a physical register and the dependents woken up. Recovery from any kind of mispredict can be handled similar to a branch mispredict. This might entail more checkpointed rename state [27]. If that is a concern, given the relatively infrequent occurrence of a mispredict, it could even be treated like an exception. The confidence thresholds can also be varied to lower the number of mispredicts.

It is worth reiterating that our goal here is not to evaluate highly complex and highly accurate prediction techniques. By demonstrating that the least complex predictors outperform cache decentralization, we have identified the least complex approach to a high performance cache organization. Our results lead us to believe that more attention should be focused on the design of high performance predictors than on cache decentralization strategies.

5.4 Sensitivity Analysis

In this subsection, we verify our results over a broader range of processor parameters. First, we examine the effect of different interconnect latencies. In the results so far, we have assumed that the two furthest clusters are separated by a minimum latency of 10 cycles. We evaluate the effect of *cluster prefetch* on two more interconnects: (i) Fast: communication on the crossbar, within a set of four clusters takes a single cycle and each hop on the ring interconnect takes two cycles. (ii) Slow: communication to and from the crossbar takes two cycles each, and each hop on the ring interconnect takes eight cycles. Figure 6 shows IPCs for these two interconnects without and with *cluster prefetch*. Performance improvements are less significant for the Fast interconnect as communication delays are less of a bottleneck (overall improvement of 11%). On the Slow interconnect, communication delays are a bottleneck on most of the programs and an overall improvement of 37% is observed. Impressive speedups are seen in *art* (217%), *equake* (42%), *fma3d* (10%), *lucas* (26%), *mesa* (32%), and *mgrid* (10%).

We also evaluated the effect of larger and smaller clusters on *cluster prefetch*. Similar result trends as before were observed. When using only 20 registers (int and fp, each) and 10 issue queue entries (int and fp, each), *cluster prefetch* yielded an overall 21% improvement. When using twice as many resources in each cluster, we continued to see an overall 22% IPC improvement. While our earlier experiments employed large 64K-entry predictors, we also evaluated the use of smaller predictors. When using predictors with 1K entries, overall performance dropped by less than 1%. We estimate that each entry requires roughly 18 bytes of storage, resulting in about 18KB worth of transistor overhead.

Finally, to make our evaluation more comprehensive, we examine the effect of *cluster prefetch* on 10 SPEC-Int programs[3]. As mentioned before, a billion-transistor architecture is best suited for programs with high parallelism or for multi-threaded workloads. By examining the behavior of SPEC-Int programs, we hope to get a flavor for the effects of *cluster prefetch* on less regular codes that might be representative of a multi-threaded workload. As expected, the performance improvements from *cluster prefetch* are not as impressive. Half the programs (*bzip, eon, gap, gzip, vortex*) yielded improvements of less than 2%, and *crafty*, *gcc*, and *vpr* yielded improvements of 3-4%. Only *twolf* and *parser* exhibited noticeable speedups (7% and 9%, respectively). The primary reason for this reduced benefit in SPEC-Int is that only 38% of all committed loads and stores have their addresses correctly predicted, while the same number in SPEC-FP is a much healthier 66%. For *cluster prefetch* to yield significant speedups in irregular code, more complex address predictors would have to be implemented.

6. RELATED WORK

A number of recent proposals [2, 3, 5, 6, 11, 29, 38] on dynamically scheduled clustered processors have advanced the state-of-the-art and many of them have been incorporated in our study, as described in Section 2. Some of these studies examine scalability issues for clustered designs. Aggarwal and Franklin [2, 3] design instruction steering algorithms and hierarchical interconnects for processors with up to 12 clusters. Parcerisa *et al.* [29] propose point-to-point interconnect designs for systems with up to eight clusters. Balasubramonian *et al.* [5] demonstrate that performance does not always scale up as the number of clusters is increased and propose adaptation algorithms to identify optimal trade-off points for low-ILP programs. Most studies assume centralized data caches and often ignore the cost of communication to this centralized resource. Zyuban and Kogge [38] incorporate a decentralized data cache in their study and Balasubramonian *et al.* [5] evaluate their results for both centralized and decentralized caches. A recent study by Racunas and Patt examines a distributed cache implementation for a clustered processor [30]. Consistent with our results, they show that decentralization results in overall improvements of less than 5%. Ours is the first study that contrasts multiple approaches to designing a complexity-effective cache for dynamically scheduled clustered processors.

Similar bottlenecks have also been studied in the domain of clustered VLIW processors. Sanchez and Gonzalez [33] propose the multiVLIW, which has multiple data cache banks to serve the clusters. A snooping cache coherence protocol ensures correctness when dealing with replicated cache blocks. Gibert *et al.* [15] use an interleaved distributed cache that eliminates the need for complex cache coherence protocols. To allow memory parallelism, the compiler has to resolve memory dependences. The authors introduce *Attraction Buffers* to prefetch spatially contiguous data. Thus, some data is replicated, and the compiler is responsible for ensuring memory correctness. In recent work [16], the same authors advocate the use of a centralized cache with compiler-managed L0 buffers to reduce communication latency. The L0 is used to cache "critical" data and "non-critical" data is mapped to the slower centralized L1 cache. In contrast to these static approaches, we rely on

[3] *Perlbmk* was not compatible with our simulator and *mcf* is too memory bound to be affected by on-chip optimizations.

dynamic techniques to speculate across unresolved memory dependences and employ address prediction to prefetch data into a cluster. As is typical with dynamic analysis, our approach entails more hardware, but can exploit run-time information to potentially perform better. However, it is interesting to note that both approaches advocate the use of a centralized cache combined with a prefetch mechanism as a low-complexity and high performance alternative. The RAW machine [7] is another example of a compiler-managed distributed on-chip cache.

Parcerisa and Gonzalez [28] evaluate the use of register value prediction to free up register dependences and hide the latency of register communication. In this paper, we employ memory address prediction to hide cache communication latency and free up memory dependences.

Many prior bodies of work have investigated the use of address prediction [4, 8, 9, 12, 17, 23, 32, 34] and memory dependence speculation [13, 22, 24] to reduce load latency. This is the first evaluation of some of these techniques in the context of a dynamically scheduled clustered processor with the goal of reducing on-chip wire delays. This is also the first comparison between prediction techniques and cache decentralization. Our results illustrate that simple implementations that would have yielded little improvement in a conventional processor can have a huge impact in reducing the effect of wire delays in future processors. We demonstrate that these different techniques interact synergistically, and the comparison with the distributed data cache suggests that address prediction and memory dependence speculation are more promising lines of research than data cache decentralization.

7. CONCLUSIONS

The paper examines data cache design issues in a highly-clustered and wire-limited microprocessor of the future. We observe that communication delays slow down the execution speeds of dependence chains, increasing the latency for effective address computation. Load latency dramatically increases because it takes a long time to compute the address, transfer it to an LSQ, wait for store addresses to determine memory dependences, and then transfer the data back to the cluster. This is especially true for a centralized data cache organization.

Our evaluations reveal that decentralizing the data cache is not an effective solution to the problem. It increases the implementation complexity and does not target the primary bottleneck – the time taken to broadcast store addresses and resolve memory dependences. Hence, we adopt a simple centralized cache organization and employ prediction techniques to deal with communication bottlenecks. We found that load address prediction and prefetch, combined with store address prediction and store-load conflict prediction yield significant speedups and outperform the decentralized cache organizations. Half the high-ILP programs in our study showed IPC improvements of between 8% and 116%. The combination of these prediction techniques is synergistic and collectively referred to as *cluster prefetch*. These apply to decentralized cache organizations as well, and we observed significant speedups in a wide variety of processor settings. Our approach also maintains low complexity and low power consumption, a primary concern for designers. The centralized cache eliminates the need for coherence traffic and *cluster prefetch* entails a modest amount of centralized transistor and logic overhead.

Thus, the contributions of the paper are:

- Demonstrating that simple prediction techniques, that yield minor improvements in a conventional processor, have a major impact in a processor constrained by communication latencies.
- Evaluating the performance potential of a decentralized organization and showing that the benefit likely does not warrant the additional implementation complexity.
- *The conclusion that address prediction and prefetch yield higher performance and entail less complexity than decentralized cache organizations. This result should help influence future research directions.*

As future work, we plan to improve the performance of our predictors and apply them to more diverse benchmark sets and to handle other long latencies on the chip, such as those for register communication and L2 access. While our proposed techniques have helped lower load latencies, they do not reduce the total bandwidth needs. In fact, the use of *cluster prefetch* removes a number of network transfers from the program critical path. Managing priorities on the network to improve performance and power remains an open problem.

8. REFERENCES

[1] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of ISCA-27*, pages 248–259, June 2000.

[2] A. Aggarwal and M. Franklin. An Empirical Study of the Scalability Aspects of Instruction Distribution Algorithms for Clustered Processors. In *Proceedings of ISPASS*, 2001.

[3] A. Aggarwal and M. Franklin. Hierarchical Interconnects for On-Chip Clustering. In *Proceedings of IPDPS*, April 2002.

[4] P. Ahuja, J. Emer, A. Klauser, and S. Mukherjee. Performance Potential of Effective Address Prediction of Load Instructions. In *Proceedings of Workshop on Memory Performance Issues (in conjunction with ISCA-28)*, June 2001.

[5] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically Managing the Communication-Parallelism Trade-Off in Future Clustered Processors. In *Proceedings of ISCA-30*, pages 275–286, June 2003.

[6] A. Baniasadi and A. Moshovos. Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduled, Superscalar Processors. In *Proceedings of MICRO-33*, pages 337–347, December 2000.

[7] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: A Compiler-Managed Memory System for Raw Machines. In *Proceedings of ISCA-26*, May 1999.

[8] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser. Correlated Load-Address Predictors. In *Proceedings of ISCA-26*, pages 54–63, May 1999.

[9] B. Black, B. Mueller, S. Postal, R. Rakvie, N. Utamaphethai, and J. Shen. Load Execution Latency Reduction. In *Proceedings of the 12th ICS*, June 1998.

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

[11] R. Canal, J. M. Parcerisa, and A. Gonzalez. Dynamic Cluster Assignment Mechanisms. In *Proceedings of HPCA-6*, pages 132–142, January 2000.

[12] T. Chen and J. Baer. Effective Hardware Based Data Prefetching for High Performance Processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[13] G. Chrysos and J. Emer. Memory Dependence Prediction Using Store Sets. In *Proceedings of ISCA-25*, June 1998.

[14] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing Cycle Time through Partitioning. In *Proceedings of MICRO-30*, pages 149–159, December 1997.

[15] E. Gibert, J. Sanchez, and A. Gonzalez. Effective Instruction Scheduling Techniques for an Interleaved Cache Clustered VLIW Processor. In *Proceedings of MICRO-35*, pages 123–133, November 2002.

[16] E. Gibert, J. Sanchez, and A. Gonzalez. Flexible Compiler-Managed L0 Buffers for Clustered VLIW Processors. In *Proceedings of MICRO-36*, December 2003.

[17] J. Gonzalez and A. Gonzalez. Speculative Execution via Address Prediction and Data Prefetching. In *Proceedings of the 11th ICS*, pages 196–203, July 1997.

[18] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Q1, 2001.

[19] U. Kapasi, W. Dally, S. Rixner, J. Owens, and B. Khailany. The Imagine Stream Processor. In *Proceedings of ICCD*, September 2002.

[20] S. Keckler and W. Dally. Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism. In *Proceedings of ISCA-19*, pages 202–213, May 1992.

[21] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[22] R. Kessler, E. McLellan, and D. Webb. The Alpha 21264 Microprocessor Architecture. In *Proceedings of ICCD*, 1998.

[23] M. Lipasti, C. Wilkerson, and J. Shen. Value Locality and Load Value Prediction. In *Proceedings of ASPLOS-VIII*, pages 138–147, October 1996.

[24] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *Proceedings of ISCA-24*, May 1997.

[25] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler. A Design Space Evaluation of Grid Processor Architectures. In *Proceedings of MICRO-34*, pages 40–51, December 2001.

[26] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The Case for a Single-Chip Multiprocessor. In *Proceedings of ASPLOS-VII*, October 1996.

[27] S. Palacharla, N. Jouppi, and J. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of ISCA-24*, pages 206–218, June 1997.

[28] J.-M. Parcerisa and A. Gonzalez. Reducing Wire Delay Penalty through Value Prediction. In *Proceedings of MICRO-33*, pages 317–326, December 2000.

[29] J.-M. Parcerisa, J. Sahuquillo, A. Gonzalez, and J. Duato. Effi cient Interconnects for Clustered Microarchitectures. In *Proceedings of PACT*, September 2002.

[30] P. Racunas and Y. Patt. Partitioned First-Level Cache Design for Clustered Microarchitectures. In *Proceedings of ICS-17*, June 2003.

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

[32] G. Reinman and B. Calder. Predictive Techniques for Aggressive Load Speculation. In *Proceedings of MICRO-31*, December 1998.

[33] J. Sanchez and A. Gonzalez. Modulo Scheduling for a Fully-Distributed Clustered VLIW Architecture. In *Proceedings of MICRO-33*, pages 124–133, December 2000.

[34] Y. Sazeides, S. Vassiliadis, and J. Smith. The Performance Potential of Data Dependence Speculation and Collapsing. In *Proceedings of MICRO-29*, pages 238–247, Dec 1996.

[35] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. Technical Report TN-2001/2, Compaq Western Research Laboratory, August 2001.

[36] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. Power4 System Microarchitecture. Technical report, Technical White Paper, IBM, October 2001.

[37] E. Tune, D. Liang, D. Tullsen, and B. Calder. Dynamic Prediction of Critical Path Instructions. In *Proceedings of HPCA-7*, pages 185–196, January 2001.

[38] V. Zyuban and P. Kogge. Inherently Lower-Power High-Performance Superscalar Architectures. *IEEE Transactions on Computers*, March 2001.