Dynamically Managing the Communication-Parallelism Trade-off in Future Clustered Processors *

Rajeev Balasubramonian†, Sandhya Dwarkadas†, and David H. Albonesi‡
† Department of Computer Science, ‡ Department of Electrical and Computer Engineering
University of Rochester, Rochester, NY 14627

Abstract

Clustered microarchitectures are an attractive alternative to large monolithic superscalar designs due to their potential for higher clock rates in the face of increasingly wire-delay-constrained process technologies. As increasing transistor counts allow an increase in the number of clusters, thereby allowing more aggressive use of instruction-level parallelism (ILP), the inter-cluster communication increases as data values get spread across a wider area. As a result of the emergence of this trade-off between communication and parallelism, a subset of the total on-chip clusters is optimal for performance. To match the hardware to the application's needs, we use a robust algorithm to dynamically tune the clustered architecture. The algorithm, which is based on program metrics gathered at periodic intervals, achieves an 11% performance improvement on average over the best statically defined architecture. We also show that the use of additional hardware and reconfiguration at basic block boundaries can achieve average improvements of 15%. Our results demonstrate that reconfiguration provides an effective solution to the communication and parallelism trade-off inherent in the communication-bound processors of the future.

1. Introduction

The extraction of large amounts of instruction-level parallelism (ILP) from common applications on modern processors requires the use of many functional units and large on-chip structures such as issue queues, register files, caches, and branch predictors. As CMOS process technologies continue to shrink, wire delays become dominant (compared to logic delays) [1, 27, 29]. This, combined with the continuing trend towards faster clock speeds, increases the time in cycles to access regular on-chip structures (caches, register files, etc.). Not only does this degrade instructions per cycle (IPC) performance, it also presents various design problems in breaking up the access into multiple pipeline stages. In spite of the growing numbers of transistors available to architects, it is becoming increasingly difficult to design large monolithic structures that aid ILP extraction without increasing design complexity, compromising clock speed, and limiting scalability in future process technologies.

A potential solution to these design challenges is a *clustered microarchitecture* [17, 29] in which the key processor resources are distributed across multiple clusters, each of which contains a subset of the issue queues, register files, and the functional units. In such a design, at the time of instruction rename, each instruction is steered into one of the clusters. As a result of decreasing the size and bandwidth requirements of the issue queues and register files, the access times of these cycle-time critical structures are greatly reduced, thereby permitting a faster clock. The simplification of these structures also reduces their design complexity.

An attractive feature of a clustered microarchitecture is the reduced design effort in producing successive generations of a processor. Not only is the design of a single cluster greatly simplified, but once a single cluster core has been designed, more of these cores can be put into the processor for a low design cost (including increasing front-end bandwidth) as the transistor budget increases. Adding more clusters could potentially improve IPC performance because each program has more resources to work with. There is little effect if any on clock speed from doing this as the implementation of each individual cluster does not change. In addition, even if the resources in a large clustered processor cannot be effectively used by a single thread, the scheduling of multiple threads on a clustered processor can significantly increase the overall instruction throughput. The relatively low design complexity and the potential to exploit thread-level parallelism make a highly-clustered processor in the billion transistor era an extremely attractive option.

The primary disadvantage of clustered microarchitectures is their reduced IPC compared to a monolithic design with identical resources. Although dependent instructions within a single cluster can issue in successive cycles, extra inter-cluster bypass delays prevent dependent instructions that lie in different clusters from issuing in successive cycles. While monolithic processors might use a potentially much slower clock to allow a single-cycle bypass among all functional units, a clustered processor allows a faster clock, thereby introducing additional latencies in cycles between some of the functional units. The clustered design is a vi-

*This work was supported in part by NSF grants EIA–0080124, CCR–9811929, CCR–9988361, CCR–0219848, and ECS–0225413; by DARPA/ITO under AFRL contract F29601-00-K-0182; by an IBM Faculty Partnership Award; by the U.S. Department of Energy Office of Inertial Confinement Fusion under Cooperative Agreement No. DE-FC03-92SF19460; and by external research and/or equipment grants from Intel, IBM, and DEC/Compaq.

able option only if the IPC degradation does not offset the clock speed improvement.

Modern processors like the Alpha 21264 [24] at 0.35μ technology already employ a limited clustered design, wherein the integer domain, for example, is split into two clusters. A number of recent studies [2, 8, 11, 12, 17] have explored the design of heuristics to steer instructions to clusters. Despite these advances, the results from these studies will likely need to be reconsidered in the near future for the following reasons:

- Due to the growing dominance of wire delays [27, 29] and the trend of increasing clock speeds, the resources in each cluster core will need to be significantly reduced relative to those assumed in prior studies.
- There will be more clusters on the die than assumed in prior studies due to larger transistor budgets and the potential for exploiting thread-level parallelism [36].
- The number of cycles to communicate data between the furthest two clusters will increase due to the wire delay problem [1]. Furthermore, communication delays will be heterogeneous, varying according to the position of the producer and consumer nodes.
- The data cache will need to be distributed among clusters, unlike the centralized cache assumed by most prior studies, due to increased interconnect costs and the desire to scale the cache commensurately with other cluster resources.

While the use of a large number of clusters could greatly boost overall throughput for a multi-threaded workload, its impact on the performance of a single-threaded program is not as evident. The cumulative effect of the above trends is that clustered processors will be much more *communication bound* than assumed in prior models.

As the number of clusters on the chip increases, the number of resources available to the thread also increases, supporting a larger window of in-flight instructions and thereby allowing more distant instruction-level parallelism (ILP) to be exploited. At the same time, the various instructions and data of the program get distributed over a larger on-chip space. If data has to be communicated across the various clusters frequently, the performance penalty from this increased communication can offset any benefit derived from the parallelism exploited by additional resources.

In this paper, we present and evaluate a dynamically tunable clustered architecture that attempts to optimize the communication-parallelism trade-off for improved single-threaded performance in the face of the above trends. The balance is effected by employing only a subset of the total number of available clusters for the thread. Our results show that the performance trend as a function of the number of clusters varies across different programs depending on the degree of distant ILP present in them. This motivates the need for dynamic algorithms that identify the optimal number of clusters for any program phase and match the hardware to the program's requirements. We present algorithms that vary the number of active clusters at any program point and show that a simple algorithm that looks at performance history over the past few intervals often yields most of the available performance improvements. However, such an algorithm misses fine-grained opportunities for reconfiguration, and we present alternative techniques that invest more hardware in an attempt to target these missed opportunities. The simple interval-based algorithm provides overall improvements of 11%, while the fine-grained techniques are able to provide 15% improvements over the best static organization.

Disabling a subset of the clusters for a given program phase in order to improve single-threaded performance has other favorable implications. Entire clusters can turn off their supply voltage, thereby greatly saving on leakage energy, a technique that would not have been possible in a monolithic processor. Alternatively, these clusters can be used by (partitioned among) other threads, thereby simultaneously achieving the goals of optimal single and multi-threaded throughput.

The rest of the paper is organized as follows. Section 2 describes the clustered microarchitecture and Section 3 describes our simulation infrastructure. Section 4 develops and evaluates our algorithms for the run-time allocation of clusters to each program phase for a centralized cache. Section 5 summarizes their performance for a decentralized cache model. In Section 6, we evaluate the sensitivity of the results to various processor parameters. We describe related work in Section 7 and conclude in Section 8.

2. The Base Clustered Processor Architecture

We start by describing a baseline clustered processor model that has been commonly used in earlier studies [2, 8, 11, 12, 17]. Such a model with four clusters is shown in Figure 1. The branch predictor and instruction cache are centralized structures, just as in a conventional processor. At the time of register renaming, each instruction gets assigned to a specific cluster. Each cluster has its own issue queue, register file, a set of functional units, and its own local bypass network. Bypassing of results within a cluster does not take additional cycles (in other words, dependent instructions in the same cluster can issue in successive cycles). However, if the consuming instruction is not in the same cluster as the producer, it has to wait additional cycles until the result is communicated across the two clusters.

A conventional clustered processor [2, 8, 11, 12, 17] distributes only the register file, issue queue, and the functional units among the clusters. The data cache is centrally located. An alternative organization [40] distributes the cache

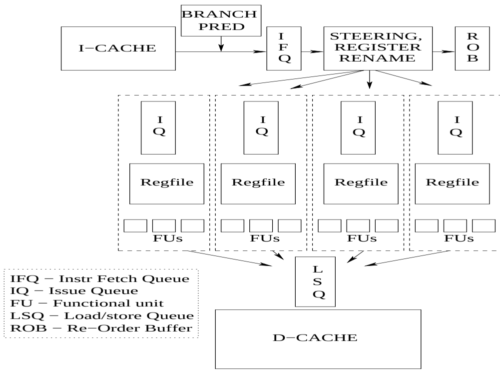

Figure 1. The base clustered processor (4 clusters) with the centralized cache.

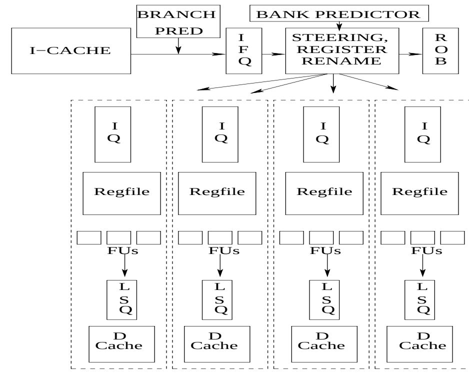

Figure 2. The clustered processor (4 clusters) with the decentralized cache.

among the clusters, thereby making the design more scalable, but also increasing the implementation complexity. Since both organizations are attractive design options, we evaluate the effect of dynamic tuning on both organizations.

2.1. The Centralized Cache

In the traditional clustered designs, once loads and stores are ready, they are inserted into a centralized load-store queue (LSQ) (Figure 1). From here, stores are sent to the centralized L1 cache when they commit and loads are issued when they are known to not conflict with earlier stores. The LSQ is centralized because a load in any cluster could conflict with an earlier store from any of the other clusters.

For the aggressive processor models that we are studying, the cache has to service a number of requests every cycle. An efficient way to implement a high bandwidth cache is to make it word-interleaved. For a 4-way word-interleaved cache, the data array is split into four banks and each bank can service one request every cycle. Data with word addresses of the form 4N are stored in bank 0, of the form 4N+1 are stored in bank 1, and so on. Such an organization supports a maximum bandwidth of four and helps minimize conflicts to a bank.

In a processor with a centralized cache, the load latency depends on the distance between the centralized cache and the cluster issuing the load. In our study, we assume that the centralized LSQ and cache are co-located with cluster 1. Hence, a load issuing from cluster 1 does not experience any communication cost. A load issuing from cluster 2 takes one cycle to send the address to the LSQ and cache and another cycle to get the data back (assuming that each hop between clusters takes a cycle). Similarly, cluster 3 experiences a total communication cost of four cycles for each load. This is in addition to the few cycles required to perform the cache RAM look-up.

Steering Heuristics: A clustered design allows a faster clock, but incurs a noticeable IPC degradation because of inter-cluster communication and load imbalance. Minimizing these penalties with smart instruction steering has been the focus of many recent studies [2, 8, 11, 12, 13, 17]. We use an effective steering heuristic [11] that steers an instruction (and its destination register) to the cluster that produces most of its operands. In the event of a tie or under circumstances where an imbalance in issue queue occupancy is seen, instructions are steered to the least loaded cluster. By picking an appropriate threshold to detect load imbalance, such an algorithm can also approximate other proposed steering heuristics like Mod_N and $First\_Fit$ [8]. The former minimizes load imbalance by steering N instructions to one cluster, then steering to its neighbor. The latter minimizes communication by filling up one cluster before steering instructions to its neighbor. We empirically determined the optimal threshold value for load balance. Further, our steering heuristic also uses a criticality predictor [18, 37] to give a higher priority to the cluster that produces the critical source operand. Thus, our heuristic represents the state-of-the-art in steering mechanisms.

2.2. The Decentralized Cache

In a highly clustered processor, the centralized cache can be a major bottleneck as it has to support a high bandwidth, and its average distance to the requesting clusters increases. Hence, a distributed cache model [40] represents an attractive design option.

For an N-cluster system, we assume that the L1 cache is broken into N word-interleaved banks. Each bank is associated with its own cluster. The LSQ is also split across the different clusters. The example in Figure 2 shows an organization with four clusters. Because they are word-interleaved, the various banks cache mutually exclusive data and do not require any cache coherence protocol between them. The goal of the steering mechanism is to steer a load or store to the cluster that caches the corresponding memory address. We discuss the additional steering complexities arising from the distributed nature of the cache in Section 5.

The L2 cache continues to be co-located with cluster 1 and a miss in any of the L1 cache banks other than that associated with this cluster incurs additional latency depending

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 mispredict penalty	at least 12 cycles
Fetch width	8 (across up to two basic blocks)
Dispatch and 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)
Re-order Buffer (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
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.

on the number of hops.

2.3. Interconnects

As process technologies shrink and the number of clusters is increased, attention must be paid to the communication delays and interconnect topology between clusters. Cross-cluster communication occurs at the front-end as well as when communicating register values across clusters or when accessing the cache. Since the former occurs in every cycle, we assume a separate network for this purpose and model non-uniform dispatch latencies as well as the additional latency in communicating a branch mispredict back to the front-end. Since the latter two (cache and register-to-register communication) involve data transfer to/from registers, we assume that the same (separate) network is used.

In our study, we focus on a ring interconnect because of its low implementation complexity. Each cluster is directly connected to two other clusters. We assume two unidirectional rings, implying that a 16-cluster system has 32 total links (allowing 32 total transfers in a cycle), with the maximum number of hops between any two nodes being 8.

In a later section, as part of our sensitivity analysis, we also show results for a grid interconnect, which has a higher implementation cost but higher performance. The clusters are laid out in a two-dimensional array. Each cluster is directly connected to up to four other clusters. For 16 clusters, there are 48 total links, with the maximum number of hops being 6, thus reducing the overall communication cost.

3. Simulation Methodology

3.1. Simulator Parameters

Our simulator is based on Simplescalar-3.0 [9] for the Alpha AXP instruction set. The simulator has been modified to represent a microarchitecture resembling the Alpha 21264 [24]. The register update unit (RUU) is decomposed into issue queues, physical register files, and the reorder buffer (ROB). The issue queue and the physical register file are further split into integer and floating-point. Thus, each cluster in our study is itself decomposed into an integer and floating-point cluster. The memory hierarchy is also modeled in detail (including word-interleaved access, bus and port contention, writeback buffers, etc).

Parameter	Centralized cache	Decentralized cache	
		each cluster	total
Cache size	32 KB	16 KB	16N KB
Set-associativity	2-way	2-way	2-way
Line size	32 bytes	8 bytes	8N bytes
Bandwidth	4 words/cycle	1 word/cycle	N words/cycle
RAM look-up time	6 cycles	4 cycles	4 cycles
LSQ size	15N	15	15N

Table 2. Cache parameters for the centralized and decentralized caches. All the caches are word interleaved. N is the number of clusters.

This base processor structure was modified to model the clustered microarchitecture. To represent a wire-delay constrained processor at future technologies, each cluster core was assumed to have one functional unit of each type, 30 physical registers (int and fp, each), and 15 issue queue entries (int and fp, each). As many instructions can issue in a cycle as the number of available functional units. We assume that each hop on the interconnect takes a single cycle. While we did not model a trace cache, we assumed that instructions could be fetched from up to two basic blocks at a time. The important simulation parameters are summarized in Table 1.

The number of resources in each cluster and the latency for each hop on the interconnect are critical parameters in such a study as they determine the amount and cost of inter-cluster communication. These parameters are highly technology, layout, and design-dependent, and determining them is beyond the scope of this study. Our results include a sensitivity analysis to see how the results change as our assumptions on the number of registers, issue queue entries, functional units, and cycles per hop are varied.

Our study focuses on wire-limited technologies of the future and we pick latencies according to projections for 0.035μ. We used CACTI-3.0 [34] to estimate access times for the cache organizations. We used the methodology in [1] to estimate clock speeds and memory latencies, following SIA roadmap projections [5]. With Simplescalar, we simulated cache organizations with different size and port parameters (and hence different latencies) to determine the best base cases. These parameters are summarized in Table 2. The centralized cache yielded best performance for a 4-way word-interleaved 32KB cache. Such a cache has a bandwidth of four accesses per cycle and an access time of six cycles. The best decentralized cache organization has a single-ported four-cycle 16KB bank in each cluster.

3.2. Benchmark Set

As a benchmark set, we used four SPEC2k Integer programs, three SPEC2k FP programs, and two programs from

Benchmark	Input dataset	Simulation window	Base IPC	Mispred branch interval
cjpeg (Mediabench)	testimg	150M-250M	2.06	82
crafty (SPEC2k Int)	ref	2000M-2200M	1.85	118
djpeg (Mediabench)	testimg	30M-180M	4.07	249
galgel (SPEC2k FP)	ref	2000M-2300M	3.43	88
gzip (SPEC2k Int)	ref	2000M-2100M	1.83	87
mgrid (SPEC2k FP)	ref	2000M-2050M	2.28	8977
parser (SPEC2k Int)	ref	2000M-2100M	1.42	88
swim (SPEC2k FP)	ref	2000M-2050M	1.67	22600
vpr (SPEC2k Int)	ref	2000M-2100M	1.20	171

Table 3. Benchmark description. Baseline IPC is for a monolithic processor with as many resources as the 16-cluster system. "Mispred branch interval" is the number of instrs before a branch mispredict is encountered.

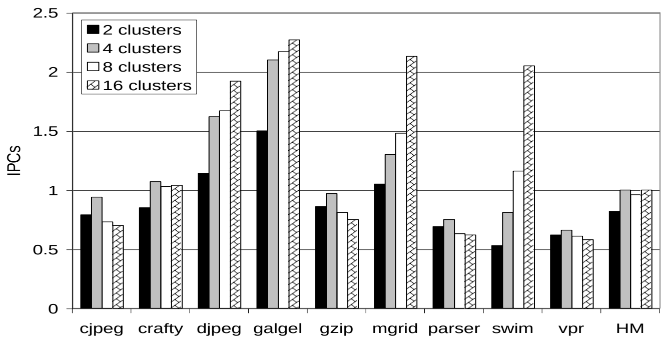

Figure 3. IPCs for fixed cluster organizations with 2, 4, 8, and 16 clusters.

the UCLA Mediabench [25]. The details on these programs are listed in Table 3. The programs represent a mix of various program types, including high and low IPC codes, and those limited by memory, branch mispredictions, etc. Most of these programs were fast forwarded through the first two billion instructions and simulated in detail to warm the various processor structures before measurements were taken. While we are simulating an aggressive processor model, not all our benchmark programs have a high IPC. Note that an aggressive processor design is motivated by the need to run high IPC codes and by the need to support multiple threads. In both cases, the quick completion of a single low-IPC thread is still important – hence the need to include such programs in the benchmark set.

4. The Dynamically Tunable Clustered Design

For brevity, we focus our initial analysis on the 16-cluster model with the centralized cache and the ring interconnect. Figure 3 shows the effect of statically using a fixed subset of clusters for a program. Increasing the number of clusters increases the average distance of a load/store instruction from the centralized cache and the worst-case inter-cluster bypass delay, thereby greatly affecting the overall communication cost. Assuming zero inter-cluster communication cost for loads and stores improved performance by 31%, while assuming zero cost for register-to-register communication improved performance by 11%, indicating that increased load/store latency dominates the communication overhead. This latency could be reduced by steering load/store instructions to the cluster closest to the cache, but this would increase load imbalance and register communication. The average latency for inter-cluster register communication in the 16-cluster system was 4.1 cycles. At the same time, using more clusters also provides the program with more functional units, registers, and issue queue entries, thus allowing it to dispatch a larger window of in-flight instructions. Depending on which of these two conflicting forces dominates, performance either improves or worsens as the number of clusters is increased. Programs with distant ILP, like *djpeg* (JPEG decoding from Mediabench), *swim*, *mgrid*, and *galgel* (loop-based floating-point programs from SPEC2K) benefit from using many resources. On the other hand, most integer programs with low branch prediction accuracies can not exploit a large window of in-flight instructions. Hence, increasing the resources only degrades performance because of the additional communication cost. This is a phenomenon hitherto unobserved in a clustered processor (partly because very few studies have looked at more than four clusters and partly because earlier studies assumed no communication cost in accessing a centralized cache).

Our goal is to tune the hardware to the program's requirements by dynamically allocating clusters to the program. This can be very trivially achieved by modifying the steering heuristic to disallow instruction dispatch to the disabled clusters. In other words, disabling is equivalent to not assigning any new instructions to the cluster. Instructions already assigned to the disabled clusters are allowed to complete, resulting in a natural draining of the cluster.

4.1. Consistency of Behavior Across Intervals

Various recent works [4, 6, 7, 10, 16, 19, 20, 22, 31, 38] have proposed run-time algorithms for the dynamic tuning of hardware to a program phase's requirements. Most of these techniques use an interval-based algorithm, where measurements over the last few intervals dictate the choice of configuration over subsequent intervals, where an interval is a pre-specified number of committed instructions. Our dynamic configuration selection mechanism is based on earlier proposals [7, 16]. At the start of each program phase, we run each configuration option for an interval and record the IPCs. We then pick the configuration with the highest IPC and use it until the next phase change is detected. Such a mechanism is heavily reliant on the program's ability to sustain uniform performance over a number of intervals. We found that floating-point programs generally show this behavior, while the integer programs show a lot more variability. While earlier studies have assumed fixed interval lengths, we found that this would result in

Benchmark	Minimum acceptable interval length and its *instability factor*	*Instability factor* for a 10K instruction interval
gzip	10K / 4%	4%
vpr	320K / 5%	14%
crafty	320K / 4%	30%
parser	40M / 5%	12%
swim	10K / 0%	0%
mgrid	10K / 0%	0%
galgel	10K / 1%	1%
cjpeg	40K / 4%	9%
djpeg	1280K / 1%	31%

Table 4. *Instability factors* for different interval lengths.

very poor performance for a number of programs. Hence, picking an appropriate interval length is fundamental to the success of a configuration selection algorithm (and can be universally applied to the configuration of other aspects of the processor in addition to the number of clusters).

To study the variability of program behavior over different intervals, we ran each of the programs for billions of instructions to generate a trace of various statistics at regular 10K instruction intervals. We used three metrics to define a program phase - IPC, branch frequency, and frequency of memory references. At the start of each program phase, the statistics collected during the first interval were used as reference. For each ensuing interval, if the three metrics for that interval were similar to the reference points, the interval was termed 'stable'. If any of the three metrics was significantly different, we declared the interval as 'unstable' and began a new program phase. This analysis was done for many interval lengths. The *instability factor* for an interval length is the percentage of intervals that were considered 'unstable', *i.e.*, the frequency of the occurrence of a phase change. In our study, we found that it was sufficient to only explore a limited subset of the possible configurations (2, 4, 8, and 16 clusters) as they covered most of the interesting cases. An *instability factor* of 5% ensures that less than 15% of the intervals are in sub-optimal configurations.

Table 4 shows the smallest interval length that affords an acceptable *instability factor* of less than 5% for each of our programs. As can be seen, the interval lengths that emerge as the best vary from 10K to 40M. We also show the *instability factor* for a fixed interval length of 10K instructions. Clearly, this interval length works poorly for a number of programs and would result in quite unacceptable performance. Most programs usually show consistent behavior across intervals for a coarse enough interval length, making interval-based schemes very robust and universally applicable. Even a program like *parser*, whose behavior varies dramatically based on the input data, has a low *instability factor* for a large 40M instruction interval.

4.2. Variable-Interval Mechanism with Exploration

In order to arrive at the optimal instruction interval length at run-time, we use a simple algorithm. We start with the minimum instruction interval. If the *instability factor* is too high, we double the size of the interval and repeat this until we either experience a low *instability factor* or until we reach a pre-specified limit (say, a billion instructions). If we reach the limit, we cease to employ the selection algorithm and pick the configuration that was picked most often.

Once we pick an interval length, we need not remain at that interval length forever. The program might move from one large *macrophase* to another that might have a completely different optimal instruction interval. To deal with this, we can continue to hierarchically build phase detection algorithms. An algorithm that inspects statistics at a coarse granularity (say, every 100 billion instructions) could trigger the detection of a new *macrophase*, at which point, we would restart the selection algorithm with a 10K interval length and find the optimal interval length all over again.

For completeness, in Figure 4, we describe our algorithm that selects the interval length, detects phases, and selects the best configuration at run-time. At the start of a phase, the statistics collected in the first interval serve as a reference point against which to compare future statistics and

```
Initializations and definitions:
interval_length = 10K;   (number of committed instrs before invoking the algo)
discontinue_algorithm = FALSE;  (if this is set, no more reconfigurations are
                                  attempted until the next macrophase)
have_reference_point = FALSE;  (the first interval in a new phase provides a
                                  reference point to compare future intervals)
significant_change_in_ipc;  (this is set if the IPC in the current interval differs
                             from that in the reference point by more than 10%)
significant_change_in_memrefs;  (this is set if the memory references in the
                                 current interval differs from the reference
                                 point by more than interval_length/100)
significant_change_in_branches;  (similar to significant_change_in_memrefs)
num_ipc_variations = 0;  (this indicates the number of times there was a
                          significant_change_in_ipc)
stable_state = FALSE;  (this is set only after all configs are explored)
num_clusters;   (the number of active clusters)
instability = 0;  (number indicating phase change frequency)

THRESH1 = THRESH2 = 5; THRESH3 = 1 billion instructions;


Inspect statistics every 100 billion instructions.
If (new macrophase)
  Initialize all variables;

If (not discontinue_algorithm)
  Execute the following after every interval_length instructions;
If (have_reference_point)
  If (significant_change_in_memrefs or significant_change_in_branches or
    (significant_change_in_ipc and num_ipc_variations > THRESH1))
    have_reference_point = stable_state = FALSE;
    num_ipc_variations = 0;
    num_clusters = 4;
    instability = instability + 2;
    if (instability > THRESH2)
      interval_length = interval_length * 2;
      instability = 0;
      if (interval_length > THRESH3)
        Pick most popular configuration; discontinue_algorithm = TRUE;
  else
     if (significant_change_in_ipc)
      if (stable_state)  num_ipc_variations = num_ipc_variations + 2;
    else
      num_ipc_variations = MAX(−2,num_ipc_variations−0.125);
    instability = instability − 0.125;
else
  have_reference_point = TRUE;
  Record branches and memrefs.

If (have_reference_point and not stable_state)
  record IPC;
  num_clusters = num_clusters * 2;
  if (num_clusters > 16)
    pick the best performing configuration;
    make its IPC the IPC_reference_point;
    stable_state = TRUE;
```

Figure 4. Run-time algorithm for dynamic selection of the number of clusters. The constant increment/decrements for num_ipc_variations and instability were chosen to allow about 5% instability. The thresholds were picked to be reasonable round numbers.

detect a phase change. The branch and memory reference frequencies are microarchitecture-independent parameters and can be used to detect phase changes even during the exploration process. After exploration, the best performing configuration is picked and its IPC is also used as a reference. A phase change is signaled if either the number of branches, the number of memory references, or the IPC differs significantly from the reference point. Occasionally, there is a slight change in IPC characteristics during an interval (perhaps caused by a burst of branch mispredicts or cache misses), after which, behavior returns to that of the previous phase. To discourage needless explorations in this scenario, we tolerate some noise in the IPC measurements (with the *num_ipc_variations* parameter). In addition, if phase changes are frequent, the *instability* variable is incremented and eventually, the interval length is doubled.

This entire process of run-time reconfiguration can be implemented in software with support from hardware event counters. A low-overhead software routine (like that used for software TLB miss handling) that inspects various hardware counters before making a decision on the subsequent configuration is invoked at every interval. The algorithm amounts to about 100 assembly instructions, only a small fraction of which are executed at each invocation. Even for the minimum interval length of 10K instructions, this amounts to an overhead of much less than 1%. Implementing the selection algorithm in software allows greater flexibility and opens up the possibility for application-specific algorithms. Algorithms at higher levels that detect changes in *macrophases* have an even lower overhead. Since the algorithm runs entirely in software, most program-specific state resides in memory as opposed to hardware registers. Hence, apart from the event counters, no additional state has to be saved and restored on a context switch.

Results. In Figure 5, the third bar illustrates the impact of using the interval-based selection mechanism with exploration at the start of each program phase. As reference points, the first two bars show the static organizations with four and 16 clusters. We see that in almost all cases, the dynamic scheme does a very good job in approximating the performance of the best static organization. For floating-point programs with little instability (*galgel, mgrid, swim*), the dynamic scheme easily matches the hardware to the program's requirements. For the integer programs, in most cases, there is an initial unstable period when the interval size is inappropriate. Consistent with our earlier analysis, the interval size is increased until it settles at one that allows an *instability factor* of less than 5%. In *parser*, the simulation interval was not long enough to allow the dynamic scheme to settle at the required 40M instruction interval.

In *djpeg*, it takes a number of intervals for the interval size to be large enough (1.28M instructions) to allow a small *instability factor*. Further, since the interval length is large,

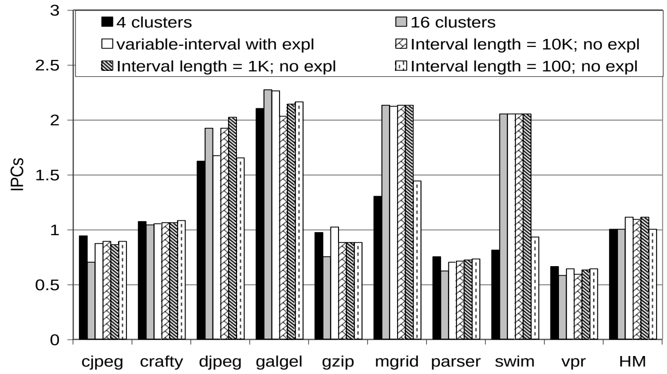

Figure 5. IPCs for the base cases and for interval-based schemes. The third bar represents the algorithm with exploration phases, while the fourth, fifth, and sixth bars represent algorithms with no exploration.

many opportunities for reconfiguration are missed. There are small phases within each interval where the ILP characteristics are different. For these two reasons, the dynamic scheme falls short of the performance of the fixed static organization with 16 clusters for *djpeg*.

In the case of *gzip*, there are a number of prolonged phases, some with distant ILP characteristics, and others with low amounts of distant ILP. Since the dynamic scheme picks the best configuration at any time, its performance is better than even the best static fixed organization.

On average, 8.3 of the 16 clusters were disabled at any time across the benchmark set. In the absence of any other workload, this produces a great savings in leakage energy, provided the supply voltage to these unused clusters can be turned off. Likewise, for a multi-threaded workload, even after optimizing single-thread performance, more than eight clusters still remain for use by the other threads.

Overall, the dynamic interval-based scheme with exploration performs about 11% better than the best static fixed organization. It is also very robust – it applies to every program in our benchmark set as there is usually a coarse enough interval length such that behavior across those intervals is fairly consistent. However, the downside is the inability to target relatively short phases. We experimented with smaller initial interval lengths, but found that the dynamic scheme encountered great instability at these small interval lengths, and hence, the interval lengths were increased to a larger value just as before. This is caused by the fact that measurements become noisier as the interval size is reduced and it is harder to detect the same program metrics across intervals and accordingly identify the best configuration for any phase.

4.3. Interval-Based Scheme with no Exploration

To alleviate these problems, we attempted an alternative interval-based scheme. Instead of exploring various configurations at the start of each program phase, we used a

16-cluster configuration for an interval and based on the degree of available distant ILP, we selected either a four or 16-cluster configuration for subsequent intervals until the next phase change (our earlier results indicate that these are the two most meaningful configurations and cover most cases). An instruction is marked as *distant* if it is at least 120 instructions younger than the oldest instruction in the ROB. At the time of issue, the instruction sets a bit in its ROB entry if it is *distant*. At the time of commit, this bit is used to increment the 'degree of distant ILP'. Since each cluster has 30 physical registers, four clusters are enough to support about 120 in-flight instructions. If the number of *distant* instructions issued in an interval exceeds a certain threshold, it indicates that 16 clusters would be required to exploit the available distant ILP. In our experiments, we use a threshold value of 160 for an interval length of 1000. Because there is no exploration phase, the hardware reacts quickly to a program phase change and reconfiguration at a finer granularity becomes meaningful. Hence, we focus on small fixed instruction intervals and do not attempt to increase the interval length at run-time. However, since the decision is based on program metrics instead of exploration, some accuracy is compromised. Further, the smaller the interval length, the faster the reaction to a phase change, but the noisier the measurements, resulting in some incorrect decisions.

Results. Figure 5 also shows results for such a mechanism for three different fixed interval lengths. An interval length of 1K instructions provides the best trade-off between accuracy and fast reactions to phase changes. Overall, it shows the same 11% improvement over the best static base case. However, in a program like *djpeg*, it does much better (21%) than the interval-based scheme with exploration because of its ability to target small phases with different requirements. Unfortunately, it takes a performance hit in programs like *galgel* and *gzip* because the small interval-length and the noisy measurements result in frequent phase changes and inaccurate decision-making.

One of the primary reasons for this is the fact that the basic blocks executed in successive 1000 instruction intervals are not always the same. As a result, frequent phase changes are signaled and each new phase change results in an interval with 16 clusters, to help determine the distant ILP. To alleviate this problem, we examine a fine-grain reconfiguration scheme at basic block boundaries.

4.4. Fine-Grain Reconfiguration

To allow reconfiguration at a fine granularity, we look upon every branch as a potential phase change. We need to determine if a branch is followed by a high degree of distant ILP, in which case, dispatch should continue freely, else, dispatch should be limited to only the first four clusters. Exploring various configurations is not a feasible option as there are likely to be many neighboring branches in different stages of exploration resulting in noisy measurements for each branch. Hence, until we have enough information, we assume dispatch to 16 clusters and compute the distant ILP characteristics following every branch. This is used to update a *reconfiguration table* so that when the same branch is later encountered, it is able to pick the right number of clusters. If we encounter a branch with no entry in the table, we assume a 16-cluster organization so that we can determine its degree of distant ILP.

Assuming that four clusters can support roughly 120 instructions, to determine if a branch is followed by distant ILP, we need to identify how many of the 360 committed instructions following a branch were *distant* when they issued. Accordingly, either four or 16 clusters would be appropriate. To effect this computation, we keep track of the *distant* ILP nature of the 360 last committed instructions. A single counter can be updated by the instructions entering and leaving this queue of 360 instructions so that a running count of the distant ILP can be maintained. When a branch happens to be the oldest of these 360 instructions, its degree of distant ILP is indicated by the value in the counter.

There is likely to still be some interference from neighboring branches. To make the mechanism more robust, we sample the behavior for a number of instances of the same branch before creating an entry for it in the reconfiguration table. Further, we can fine-tune the granularity of reconfiguration by attempting changes only for specific branches. For example, we found that best performance was achieved when we attempted changes for only every fifth branch. We also show results for a mechanism that attempts changes only at subroutine calls and returns. We formalize the algorithm below:

```
At every Nth branch, look up the reconfig table.
  If entry found, change to advised configuration.
  Else,           use 16 clusters.
While removing a branch from the queue of 360
committed instrs,
  If M samples of this branch have been seen,
      Do not update table.
  Else,
      Record the latest sample.
      If this is the Mth sample,
          compute the advised configuration.
      Else,
          advised configuration is 16 clusters.
```

The downside of the approach just described is the fact that initial measurements dictate future behavior. The nature of the code following a branch could change over the course of the program. It might not always be easy to detect such a change, especially if only four clusters are being used and the degree of distant ILP is not evident. To deal with this situation, we flush the reconfiguration table at periodic intervals. We found that re-constructing the table every 10M instructions resulted in negligible overheads.

Results. In Figure 6, in addition to the base cases and the interval-based scheme with exploration, we show IPCs

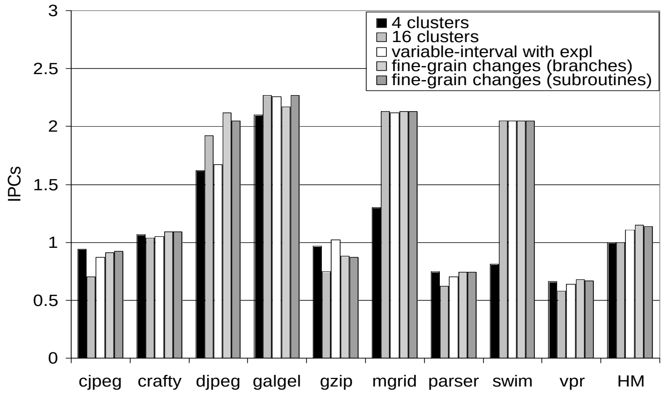

Figure 6. IPCs for the base cases, the interval-based algorithm with exploration, and two fine-grained reconfiguration schemes. The first reconfigures at every fifth branch, the second at every subroutine call and return.

for two fine-grained reconfiguration schemes. The first attempts reconfiguration at every 5th branch and creates an entry in the table after collecting 10 samples for each branch. To eliminate effects from aliasing, we use a large 16K-entry table, though, in almost all cases, a much smaller table works as well. The second scheme attempts changes at every subroutine call and return and uses three samples. The figure indicates that the ability to quickly react to phase changes results in improved performance in programs like *djpeg, cjpeg, crafty, parser,* and *vpr*. The maximum number of changes between configurations was observed for *crafty* (1.5 million). Unlike in the interval-based schemes with no exploration, instability is not caused by noisy measurements. However, *gzip* fails to match the performance achieved by the interval-based scheme. This is because the nature of the code following a branch changes over the course of the program. Hence, our policy of using initial measurements to pick a configuration for the future is not always accurate. The same behavior is observed to a lesser extent in *galgel*. Overall, the fine-grained schemes yield a 15% improvement over the base cases, compared to the 11% improvements seen with the interval-based schemes.

From these results, we conclude that interval-based schemes with exploration are easy to implement, robust, and provide most of the speedups possible. Because of their tendency to pick a coarse interval length, a number of reconfiguration opportunities are missed. Choosing a small interval length is not the solution to this because of noisy measurements across successive small intervals. To allow fine-grained reconfigurations, we pick basic block boundaries as reconfiguration points and use initial measurements to predict future behavior. Except for *gzip*, such an approach does not trade off much accuracy and the hardware is able to quickly adapt to the program's needs. However, to get this additional 4% improvement, we have to invest some non-trivial amount of hardware – a table to keep track of the predictions and logic to maintain the distant ILP metric.

5. The Decentralized Cache Model

Clustered LSQ implementation. In the decentralized cache model, if an effective address is known when a memory instruction is renamed, then it can be directed to the cluster that caches the corresponding data. However, the effective address is generally not known at rename time, requiring that we predict the bank that this memory operation is going to access. Based on this prediction, the instruction is sent to one of the clusters. Once the effective address is computed, appropriate recovery action has to be taken in the case of a bank misprediction.

If the operation is a load, recovery is simple - the effective address is sent to the correct cluster, where memory conflicts are resolved in the LSQ, data is fetched from the cache bank, and returned to the requesting cluster. If the memory operation is a store, the mis-direction could result in correctness problems. A load in a different cluster could have proceeded while being unaware of the existence of a mis-directed store to the same address. To deal with this problem, we adopt a policy similar to that in [40]. While renaming, a store whose effective address is unknown is assigned to a particular cluster (where its effective address is computed), but at the same time, a dummy slot is also created in the other clusters. Subsequent loads behind the dummy slot in other clusters are prevented from proceeding because there is an earlier store with an unresolved address that could potentially cause conflicts. Once the effective address is computed, the information is broadcast to all the clusters and the dummy slots in all the LSQs except one are removed. The broadcast increases the traffic on the interconnect for register and cache data (which we model).

Bank prediction. Earlier work by Yoaz et al. [39] had proposed the use of branch-predictor-like tables to predict the bank accessed by a load or store. In our simulations, we use a two-level bank predictor with 1024 entries in the first level and 4096 entries in the second.

Steering heuristics. In a processor with a decentralized cache, the steering heuristic has to handle three data dependences for each load or store – the two source operands and the bank that caches the data. Since the transfer of cache data involves two communications (the address and the data), performance is maximized when a load or store is steered to the cluster that is predicted to cache the corresponding data (note that unlike in the centralized cache model, doing so does not increase load imbalance as the cache is not at a single location). Even so, frequent bank mispredictions and the increased traffic from store address broadcasts seriously impact performance. Ignoring these effects improved performance by 29%. At the same time, favoring the dependence from the cache bank results in in-

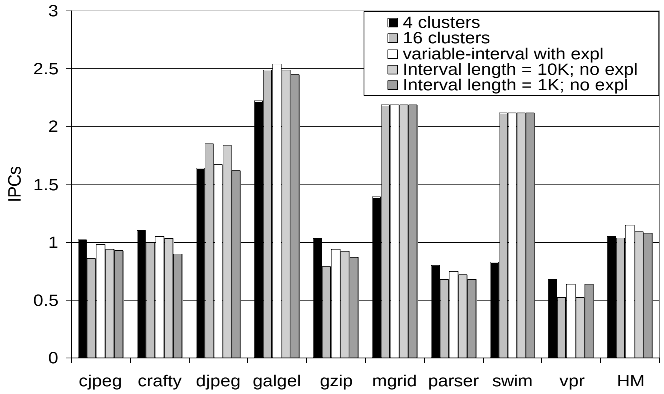

Figure 7. IPCs for dynamic interval-based mechanisms for the processor model with the decentralized cache.

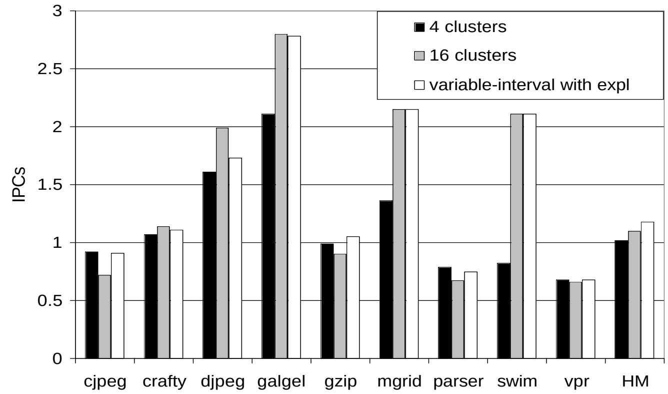

Figure 8. IPCs for the dynamic interval-based mechanism for the processor model with the grid interconnect.

creased register communication. Assuming free register communication improved performance by 27%. Thus, register and cache traffic contribute equally to the communication bottleneck in such a system.

Disabling clusters. So far, our results have assumed a clustered processor with a centralized cache. Hence, reconfiguration is only a matter of allowing the steering heuristic to dispatch to a subset of the total clusters. With a decentralized cache, each cluster has a cache bank associated with it. Data is allocated to these cache banks in a word-interleaved manner. In going from 16 to four clusters, the number of cache banks and hence, the mapping of data to physical cache lines changes. To fix this problem, the least complex solution is to stall the processor while the L1 data cache is flushed to L2. Fortunately, the bank predictor need not be flushed. With 16 clusters, the bank predictor produces a 4-bit prediction. When four clusters are used, the two lower order bits of the prediction indicate the correct bank.

Results. Because the indexing of data to physical cache locations changes, reconfiguration is not as seamless as in the centralized cache model. Every reconfiguration requires a stall of the processor and a cache flush. Hence, the fine-grained reconfiguration schemes from the earlier section do not apply. Figure 7 shows IPCs for the base cases and the interval-based mechanisms. The third bar shows the scheme with exploration and a minimum interval length of 10K instructions. The fourth and fifth bars show interval-based schemes with no exploration and the use of distant ILP metrics to pick the best configuration. The simulation parameters for the decentralized cache are summarized in Table 2. We find that the results trend is similar to that seen before for the centralized cache model. Except in the case of *djpeg*, there is no benefit from reconfiguring using shorter intervals. Overall, the interval-based scheme with exploration yielded a 10% speedup over the base cases.

Since the dynamic scheme attempts to minimize reconfigurations, cache flushes are kept to a minimum. *Vpr* encountered the maximum number of writebacks due to flushes (400K), which resulted in a 1% IPC slowdown. Overall, these flushes resulted in a 0.3% IPC degradation.

6. Sensitivity Analysis

Our results have shown that the communication-parallelism trade-off greatly affects the scalability of different programs as the number of clusters is increased for two important cache organizations. In this section, we confirm the applicability of our dynamic reconfiguration algorithms to other meaningful base cases. Some of the key parameters that affect the degree of communication and the degree of distant ILP are the choice of interconnect between the clusters, the latency of communication across a hop, the number of functional units in each cluster, and the number of instructions that can be supported by each cluster (the number of registers and issue queue entries per cluster).

Figure 8 shows the effect of using a grid interconnect as described in Section 2.3 with a centralized cache model. Because of the better connectivity, the communication is less of a bottleneck and the performance of the 16-cluster organization is 8% better than that of the 4-cluster system. For brevity, we only show results with the interval-based scheme with exploration. The trend is as seen before, but because the communication penalty is not as pronounced, the overall improvement over the best base case is only 7%. The use of fine-grained reconfiguration techniques yields qualitatively similar results as with the ring interconnect.

We also studied the sensitivity of the results to the sizes of various resources within a cluster. We studied the effect of using fewer (10 issue queue entries and 20 registers per cluster) and more resources (20 issue queue entries and 40 registers per cluster). When there are few resources per cluster, more clusters are required, on average, to exploit the available parallelism. Hence, the 16-cluster system is a favorable base case and the improvement of the interval-based dynamic mechanism relative to it is only 8%. When there are more resources per cluster, using a few clusters for low-ILP phases is highly beneficial. As a result, the improvement over the 16-cluster base is 13%. By using more functional units per cluster, our results were very similar to those in Section 4.2. Doubling the cost of communication across each hop results in a highly communication-bound 16-cluster system. By employing the dynamic mechanism and using fewer clusters for low-ILP phases, a 23% performance improvement was seen.

These results are qualitatively similar to the improvements seen with the interval-based schemes in the earlier subsections, indicating that the dynamically tunable design can help improve performance significantly across a wide range of processor parameters. Thus, the communication-parallelism trade-off and its management are likely to be important in most processors of the future.

7. Related Work

A number of proposals based on clustered processors have emerged over the past decade [3, 8, 11, 12, 13, 17, 23, 26, 28, 30, 32, 33, 35]. These differ in the kinds of resources that get allocated, the instruction steering heuristics, and the semantics for cross-cluster communication. The cache is a centralized structure in all these models. These studies assume a small number of total clusters with modest communication costs.

Cho et al. [14, 15] cluster the cache and LSQ, but not the rest of the processor. Stack and frame data are cached in a separate bank and loads and stores are steered to one of two streams early in the pipeline. Yoaz et al. [39] anticipate the importance of splitting accesses across multiple streams and propose predictors for the same.

Recently, Zyuban and Kogge [40] incorporated a clustered cache in their study on the power efficiency of a clustered processor. Our implementation of the decentralized cache closely resembles theirs. A recent study by Aggarwal and Franklin [2] explores the performance of various steering heuristics as the number of clusters scale up. Theirs is the only study that looks at as many as 12 clusters and proposes the use of a ring interconnect. They conclude that the best steering heuristic varies depending on the number of clusters and the processor model. To take this into account, each of our clustered organizations was optimized by tuning the various thresholds in our steering heuristic.

Many recent bodies of work [4, 6, 7, 10, 16, 19, 20, 22, 31, 38] have looked at hardware units with multiple configuration options and algorithms for picking an appropriate configuration at run-time. Many of these algorithms are interval-based, in that, they monitor various statistics over a fixed interval of instructions or cycles and make configuration decisions based on that information. Ours is the first proposal that identifies the importance of a variable-length instruction interval and incorporates this in the selection algorithm. We are also the first to look at fine-grained reconfiguration at branch boundaries and contrast it with interval-based schemes. Huang et al. [21] study adaptation at subroutine boundaries and also demonstrate that this can be more effective than using fixed instruction intervals.

Agarwal et al. [1] show that processors in future generations are likely to suffer from lower IPCs because of the high cost of wire delays. Ours is the first study to focus on a single process technology and examine the effects of adding more resources. The clustered processor model exposes a clear trade-off between communication and parallelism, and it readily lends itself to low-cost reconfiguration.

8. Conclusion

We have presented and evaluated the effects of shrinking process technologies and dominating wire delays on the design of future clustered processors. While increasing the number of clusters to take advantage of the increasing chip densities improves the processor's ability to support multiple threads, the performance of a single thread can be adversely affected. This is because such processors are bound by cross-cluster communication costs. These costs can tend to dominate any increased extraction of instruction-level parallelism as the processor is scaled to large numbers of clusters. We have demonstrated that dynamically choosing the number of clusters using an exploration-based approach at regular intervals is effective in optimizing the communication-parallelism trade-off for a single thread. It is applicable to almost every program and yields average performance improvements of 11% over our base architecture. In order to exploit phase changes at a fine grain, additional hardware has to be invested, allowing overall improvements of 15%. Since 8.3 clusters, on average, are disabled by the reconfiguration schemes, there is the potential to save a great deal of leakage energy in single-threaded mode. The throughput of a multi-threaded workload can also be improved by avoiding cross-thread interference by dynamically dedicating a set of clusters to each thread. We have verified the validity of our results for a number of interesting processor models, thus highlighting the importance of the management of the communication-parallelism trade-off in future processors.

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 the ISCA-27*, pages 248–259, 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] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. In *Proceedings of MICRO-31*, 1998.

[4] D. Albonesi. Dynamic IPC/clock rate optimization. *Proceedings of the 25th International Symposium on Computer Architecture*, pages 282–292, June 1998.

[5] S. I. Association. The National Technology Roadmap for Engineers. Technical report, 1999.

[6] R. I. Bahar and S. Manne. Power and Energy Reduction Via Pipeline Balancing. In *Proceedings of ISCA-28*, July 2001.

[7] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. In *Proceedings of MICRO-33*, pages 245–257, Dec 2000.

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

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

[10] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi. An Adaptive Issue Queue for Reduced Power at High Performance. In *Workshop on Power-Aware Computer Systems (PACS2000, held in conjunction with ASPLOS-IX)*, Nov 2000.

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

[12] R. Canal, J. M. Parcerisa, and A. Gonzalez. Dynamic Code Partitioning for Clustered Architectures. *International Journal of Parallel Programming*, 29(1):59–79, 2001.

[13] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned Register Files for VLIWs: A Preliminary Analysis of Trade-offs. In *Proceedings of MICRO-25*, 1992.

[14] S. Cho, P.-C. Yew, and G. Lee. Access Region Locality for High-Bandwidth Processor Memory System Design. In *Proceedings of MICRO-32*, pages 136–146, 1999.

[15] S. Cho, P.-C. Yew, and G. Lee. Decoupling Local Variable Accesses in a Wide-Issue Superscalar Processor. In *Proceedings of ISCA-26*, pages 100–110, 1999.

[16] A. Dhodapkar and J. E. Smith. Managing Multi-Configurable Hardware via Dynamic Working Set Analysis. In *Proceedings of ISCA-29*, May 2002.

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

[18] B. Fields, S. Rubin, and R. Bodik. Focusing Processor Policies via Critical-Path Prediction. In *Proceedings of ISCA-28*, July 2001.

[19] D. Folegnani and A. Gonzalez. Reducing Power Consumption of the Issue Logic. In *Workshop on Complexity-Effective Design (WCED2000, held in conjunction with ISCA-27)*, June 2000.

[20] S. Ghiasi, J. Casmira, and D. Grunwald. Using IPC Variations in Workloads with Externally Specified Rates to Reduce Power Consumption. In *Workshop on Complexity Effective Design (WCED2000, held in conjunction with ISCA-27)*, June 2000.

[21] M. Huang, J. Renau, and J. Torrellas. Positional Adaptation of Processors: Applications to Energy Reduction. In *Proceedings of ISCA-30*, June 2003.

[22] M. Huang, J. Reneau, S. Yoo, and J. Torrellas. A Framework for Dynamic Energy Efficiency and Temperature Management. In *Proceedings of MICRO-33*, pages 202–213, Dec 2000.

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

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

[25] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of MICRO-30*, pages 330–335, 1997.

[26] P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. O'Donnell, and J. Ruttenberg. The Multiflow Trace Scheduling Compiler. *Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[27] D. Matzke. Will Physical Scalability Sabotage Performance Gains? *IEEE Computer*, 30(9):37–39, Sept 1997.

[28] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler. A Design Space Evaluation of Grid Processor Architectures. In *Proceedings of MICRO-34*, Dec 2001.

[29] S. Palacharla, N. Jouppi, and J. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of ISCA-24*, 1997.

[30] J.-M. Parcerisa, J. Sahuquillo, A. Gonzalez, and J. Duato. Efficient Interconnects for Clustered Microarchitectures. In *Proceedings of PACT*, Sep 2002.

[31] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources. In *Proceedings of MICRO-34*, Dec 2001.

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

[33] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace Processors. In *Proceedings of MICRO-30*, 1997.

[34] 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.

[35] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *Proceedings of ISCA-22*, 1995.

[36] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of ISCA-22*, pages 392–403, 1995.

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

[38] S. Yang, M. Powell, B. Falsafi, K. Roy, and T. Vijaykumar. An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep Submicron High-Performance I-Caches. In *Proceedings of HPCA-7*, Jan 2001.

[39] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation Techniques for Improving Load Related Instruction Scheduling. In *Proceedings of ISCA-26*, pages 42–53, 1999.

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