A Dependency Chain Clustered Microarchitecture

Satish Narayanasamy[†] Hong Wang[‡] Perry Wang[‡] John Shen[‡] Brad Calder[†]

[†]Department of Computer Science and Engineering, University of California, San Diego
[‡]Microarchitecture Research Lab, Intel Corporation
{satish,calder}@cs.ucsd.edu {hong.wang, perry.wang, john.shen}@intel.com

Abstract

In this paper we explore a new clustering approach for reducing the complexity of wide issue in-order processors based on EPIC architectures. Complexity effectiveness is achieved by heavily clustering the pipeline from decode to commit stage without the need for any direct bypass between clusters. This is made possible by assuming support for executing compiler-constructed traces. One trace is executed at a time by executing its coarse-grained dependency chains (DCs) in different in-order clusters. Since the DCs of a trace are mutually data independent of each other they can be executed in different clusters without any direct communication between them. To execute DCs in narrower clusters without compromising ILP, a compiler algorithm that splits large DCs by duplicating instructions is proposed.

Through cycle accurate simulations we show that a DC processor with one 3-wide, one 2-wide and one 1-wide in-order pipeline, could achieve performance equivalent to a 6-wide in-order superscalar processor. Since a clustered DC microarchitecture is complexity efficient, it is amenable to higher clock frequencies and will also be easier to design and validate than a 6-wide monolithic design.

1 Introduction

In this research we investigate microarchitectural techniques to design future complexity effective high performance processor implementations based on EPIC philosophy [20]. Current processors implementing the Intel Itanium Processor Family (IPF) [16] use a monolithic 6-wide in-order pipeline. Instruction level parallelism (ILP) is exploited through aggressive optimizations in the compiler. In order to further improve the performance of IPF designs, one obvious choice is to exploit more ILP by designing out-of-order implementations for IPF, but this has serious complexity issues. Another option is to get higher frequency by designing deeper pipelines for the monolithic in-order pipeline, but this can increase both branch penalty and cache/memory latency that can hurt IPC. This paper explores a third alternative to achieve higher IPC and frequency than current IPF implementations, by using an aggressively clustered *Dependency Chain Processor* (DCP) microarchitecture that assumes compiler support for generating optimized software traces and their dependency chains.

Clustering the processor resources to design complexity efficient microarchitectures has been an important topic of research [9, 11, 3, 18, 8]. Instead of a monolithic pipeline, multiple clusters are used where each cluster has less complexity. Instructions are dispatched to different clusters with the objective of minimizing inter-cluster communication, which can incur longer latencies. Recently, dependency chain based execution has been proposed to alleviate the scheduler complexity by exploiting the inter-instruction dependency information [13]. We explore novel ways to judiciously use the above two ideas to design aggressively clustered microarchitectures for in-order processor implementations.

A DCP microarchitecture assumes support for executing traces to improve ILP [10]. Traces are generated offline in software, using instruction profiles and are much longer (even 100s of instructions) than the traces used in conventional trace caches [10]. They can be generated by an EPIC compiler and can be another means for the compiler to convey parallelism to the architecture. Alternatively, these traces can be generated using binary instrumentation tools like Ispike [15] and appended to the existing binary or can potentially even be generated on-line using a dynamic optimizer [4]. Constructing long traces and doing aggressive instruction scheduling over large regions of code, could aid in exploiting parallelism even beyond traditional out-of-order processor's instruction window size. Our simulation studies show an improvement of up to 15% in IPC by executing scheduled traces in a 6-wide in-order processor.

Each trace is partitioned into multiple mutually data independent dependency chains (DCs) - a dependency chain in a trace is a maximally connected component in the data dependency graph of the trace. Note, this does not mean that all the instructions in a dependency chain are linearly dependent on one another, since some instructions can have independent ancestors.

The DCs that we construct are much larger than the ones studied before [11] and the largest one in a trace may contain 10s of instructions. DCs from the same trace are mapped to different clusters and dispatched simultaneously to the corresponding cluster queues. As these DCs execute independently in different clusters, higher ILP can be achieved through out-of-order execution of instructions across the clusters. In addition, higher ILP is exploited within a cluster as instructions within a DC are scheduled greedily. A key advantage of this microarchitectural model is that since the DCs of a trace are mutually independent, no inter-cluster communication is needed while executing a trace. This allows aggressive clustering of processor resources from decode to the commit stage in the pipeline, which will make it possible to achieve higher clock frequencies.

This paper makes several contributions. For our experiments

we devise an approach to identify and form traces that is implementation independent. We present a characterization of DCs based on these traces. We observe that in most traces, there is one dominant DC that becomes the critical bottleneck to performance as it accounts for 50-60% of instructions in the trace. We then devise a DC splitting algorithm that facilitates the reduction of the dominant DC size by about 20% on average. This optimized trace and DC construction aids in building a core with one 3-wide, one 2-wide and one 1-wide clusters, with in-order scheduling in each cluster and no inter-cluster bypass. This achieves IPC equivalent to that of a monolithic 6-wide in-order execution core that executes scheduled traces. Our DC-based processor has the potential of achieving high overall IPC equivalent to that of an optimized in-order design executing scheduled traces, without having to pay the complexity, and power, of a wide monolithic execution core.

The rest of the paper is organized as follows. Section 2 reviews the related work and Section 3 presents our benchmark and platform methodology. Section 4, describes the trace construction algorithm. Section 5 analyses the properties of the DCs. Instruction duplication algorithm to optimize DCs is discussed in Section 6. Our microarchitecture model is proposed in Section 7 and its performance analysis is presented in Section 8. Finally, Section 9 concludes.

2 Related Work

2.1 Complexity Effective Designs

Palacharla et al., [18] identified rename, bypass network in the execution stage, and issue-wakeup logic to be the most performance critical structures in the pipeline that prohibit achieving higher clock frequencies. In addition to being amenable to higher clock frequencies, complexity effective structures could also be expected to consume less energy and will be easier to design and validate. One solution to reduce the complexity is to partition the resources into multiple smaller clusters [9].

A clustered microarchitecture however introduces overheads in the form of imbalance distribution of instructions among clusters and additional overhead due to increased latencies for inter-cluster communication. Designing an efficient cluster architecture that addresses these problems has been an important topic of research [11, 9, 18, 8, 6] and there has also been prior work on compiler optimizations to partition instructions between clusters [3]. In this work we concentrate on designing clustered in-order microarchitectures. Key advantage of our DCP architecture over previously proposed solutions is that it is heavily clustered from decode to commit stage and does not require any direct bypass network for communication between clusters. Also, since the DCs are constructed and mapped to different clusters in a compiler, there is no need for a dynamic cluster scheduling mechanism like in previous solutions [6].

2.2 Chain-Based Execution

We assume architecture support for constructing [15, 4] and executing [17] instruction traces. Nair et al. [17] discussed the advantages of executing scheduled instruction traces in an in-order processor.

There have been proposals that dynamically construct and execute small sequences of dependent instructions in order to reduce the complexity of instruction scheduler [19, 13]. Kim and Lipasti [13] aggregated a small number (about 5) of data dependent instructions into a coarser-grained macro-op which can be executed in a pipelined scheduler with a larger instruction window that has lower hardware complexity than traditional instruction schedulers. Probably the closest work to ours is the Instruction Level Distributed Processing (ILDP) microarchitecture proposed by Kim and Smith [11]. In the ILDP [11] design, a traditional dynamic instruction scheduler is substituted with a set of FIFO queues and dependent instructions are steered into the same queue, possibly with the support of dynamic binary translation [12]. All of the above prior work focuses on constructing dependency units from a small trace or instruction window where each dependency unit typically contain 3 to 5 instructions. DCP uses very coarse-grained dependency chains (DCs) comprising 10s of dependent instructions. Such DCs are constructed from long traces to facilitate efficient execution on heavily clustered microarchitectures.

2.3 Instruction Replication Algorithms

We optimize DCs to execute them on narrow wide clusters by splitting them using instruction duplication algorithm. Aggarwal et al. [1] studied a technique to dynamically replicate instructions to reduce inter-cluster communication delays. Their instruction replication is a hardware mechanism added to the instruction dispatch stage in a dynamically scheduled processor. But in this work, we do an offline analysis at the granularity of coarser dependency chains in order to split the dependency chains so that they can be efficiently executed on a heavily clustered architecture. Aleta et al., [2] use instruction replication on top of the traditional modulo scheduling algorithm to effectively reduce inter-cluster communication. However, the workloads targeted by [2] are primarily computation-intensive floating-point applications with high ILP. Complementary to [2], our study focuses on the scalar integer applications that tend to have irregular data flow patterns. Hence, our objectives and heuristics for instruction duplication are different from previously proposed algorithms [2, 1].

3 Benchmarks

Since the goal of this work is to explore future high performance complexity effective in-order designs, we base this research on Intel Itanium Processor Family (IPF) instruction set architecture [16]. The benchmarks used for this study include eight integer benchmarks chosen from the SPEC CINT2000 suite. The programs were compiled using the Intel Electron compiler for the IPF architecture. This compiler implements many state-of-the-art compiler optimization techniques such as profile-driven feedback directed optimizations, aggressive software prefetching, software pipelining, control speculation and data speculation. Predication is not used in order to examine the DCs in a more architecturally independent manner.

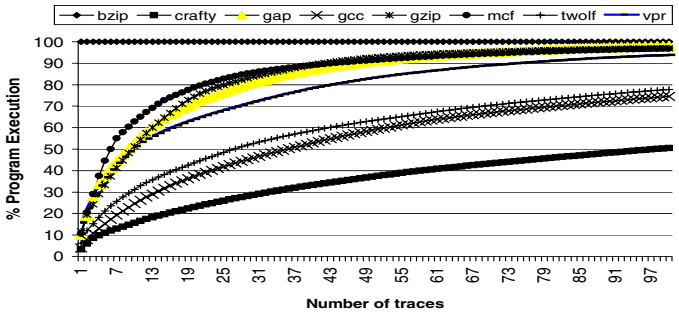

Figure 1: Cumulative Percentage of dynamic instructions covered by traces captured using modified WPP algorithm. Fewer than 100 traces cover over 80% of program execution for most programs.

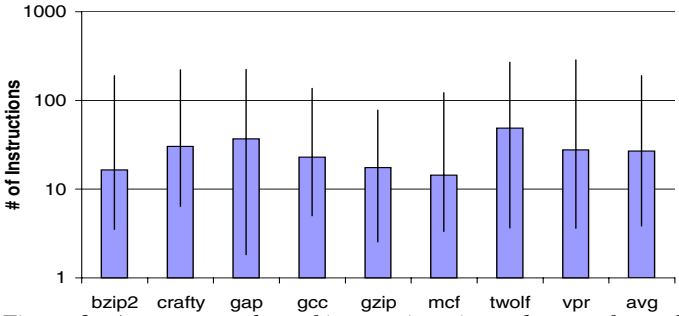

Figure 2: Average number of instructions in each trace formed using modified WPP approach. Vertical line on top of the each bar shows the maximum and minimum trace length. Y-axis is in logarithmic scale.

```
I00,       st8                 [r6]=r3
I01,       adds                r6=04h, r6
I02,       adds                r7=01h, r7
I03,       cmp4.lt.unc         p9, p8 = 32h, r7
I04, (p8)  br.cond.dpnt.few    b0 = $+0x42b3a8
I05,       ld8                 r5 = [r6]
I06,       adds                r3 = 0, r0
I07,       cmp4.lt.unc         p9, p8 = r5, r4
I08, (p8)  br.cond.dpnt.few    b0 = $+0x42b3d0
I09,       st8                 [r6]=r3
I10,       adds                r6=04h, r6
I11,       adds                r7=01h, r7
I12,       cmp4.lt.unc         p9, p8 = 32h, r7
I13, (p8)  br.cond.dpnt.few    b0 = $+0x42b3a8
```

Figure 3: A sample trace

4 DC-Trace Construction

A DCP microarchitecture assumes trace-based execution model. Long traces are constructed offline from instruction profiles and are used during performance simulation. Constructing longer traces will enable us to generate aggressive trace schedules that will help us to exploit more ILP while using in-order clusters.

4.1 Trace Formation Algorithm

We refer to the traces constructed for a DCP microarchitecture as DC-Traces. The DC-Trace construction is based on the Whole Program Path (WPP) approach proposed by Larus [14]. The instruction profile of program execution is divided into atomic blocks, where an atomic block is a sequence of instructions between the target of a taken branch and the following branch instruction. Each unique atomic block is assigned a unique identifier. As a result, the instruction trace is converted into a string of identifiers. This string is then given as input to the Sequitur algorithm [7].

Sequitur is a compression algorithm that produces a compact representation for any given input string by representing it in the form of a context free grammar. The algorithm replaces a sequence of identifiers and grammar rules that repeats more than once with a single grammar rule. The advantage of this algorithm is that the compression is done in linear time. The resultant grammar representation is a directed acyclic graph (DAG) with the grammar rules represented as intermediate nodes and the identifiers represented as leaves. A consecutive sequence of identifiers on the right hand side of a grammar rule constitutes a trace, which can be obtained by parsing the DAG.

4.2 Trace Characteristics

Figure 1 shows the coverage of the top 100 most frequently executed traces. The x-axis shows the number of traces and the y-axis depicts the cumulative percentage of program coverage of those traces. We can see that fewer than 100 traces are necessary to cover nearly 80% of dynamically executed instructions for all programs except crafty, which has a larger program footprint. For bzip, just one trace accounts for 99.9% of program execution in the simulation window that we studied. Figure 2 presents the number of instructions in a trace averaged over the 100 frequently executed traces. It can be noted that the traces that we construct on an average contain around 30 instructions but the longest trace contains about 258 instructions (for vpr). These are much longer when compared to the traces used in conventional trace processors (16 instructions) [10]. Hereafter, all analysis is done using these top 100 most frequently executed traces.

5 Characteristics of Dependency Chains

In this section we will quantify and analyze the properties of dependency chains which will aid us in designing an efficient clustered microarchitecture.

A dependency chain (DC) is defined as a maximally connected component in the data dependency graph of a trace. In this section we examine the DCs found within the DC-Traces created as described in the prior section. Our goal is to have narrow and independent DCs, so we first focus on two metrics to analyze the number of instructions within a DC and the width of the DC. Figure 3 shows a sample trace and Figure 4 shows the three DCs corresponding to that trace. Among the three DCs, DC2 is the most dominant since it contains more instructions than all others. We refer to such a DC as the 1st dominant DC (or simply as dominant DC) of a trace. The DC that has the second largest number of instructions is referred to as the 2nd dominant DC and so on. In the example shown in Figure 4, DC3 is the 2nd dominant DC of the trace.

Figure 5 shows the percentage of instructions in a trace accounted by the top 7 dominant DCs, averaged over the top 100 frequently executed traces for each of the benchmarks studied. The figure shows that 1st dominant DC in a trace contains a large number of instructions when compared to the other DCs in the trace. The 1st dominant DC in any trace accounts for about 50%-60% of instructions in the trace while the 2nd dom-

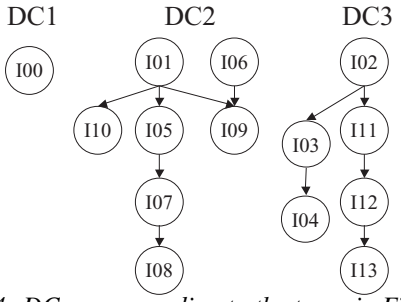

Figure 4: DC corresponding to the trace in Figure 3

inant DCs accounts for approximately 20%, and the rest of the DCs account for fewer than 10%.

We are also interested in another metric for DCs, which we call DC-Width. DC-Width of a DC is the number of instructions in the DC divided by the critical path length (the longest length of serial data dependencies) in the data dependency graph of that DC. Since all the instructions in a particular DC will be executed within a single cluster, the DC-Width metric represents the minimum issue width necessary in a cluster to exploit all of the available ILP in the DC.

Figure 6 shows the DC-Width distribution for the top 7 DCs averaged across all the top 100 frequently executed traces from all the benchmarks. For some traces, the 2nd dominant DC and other less dominant DCs may not exist at all and hence the bars for those less dominant DCs do not add up to 100%.

In 55% of the traces executed, the DC-Widths of the 1st dominant DCs are greater than 2 (in other words, in 45% of the traces executed, the DC-Widths of the 1st dominant DCs are less than 2). Also, DC-Width tends to be greater than 3 only for the 1st dominant DCs but for others it is rarely greater than two. This implies that while designing a clustered microarchitecture, we need not design clusters with uniform width. Instead, by having just one wide cluster catering to the needs of the 1st dominant DCs and multiple narrower clusters, each possibly 1 instruction wide, one can exploit most of the available ILP. This property motivates a case for designing a heterogeneous clustered microarchitecture which has several advantages, which will be discussed in Section 8.3.

Figure 7 shows the average critical path length of the top 7 dominant DCs averaged over all the traces. As expected, the critical path of the 1st dominant DCs is longer than that of the others and also the combined path length of less dominant DCs usually does not exceed that of the 1st dominant DC. Hence while the 1st dominant DC is executing in a wide cluster, all other DCs can be scheduled to execute sequentially on one or more narrow width clusters. The policy that is used to map DCs to clusters is discussed more in detail in Section 7.2.

6 Instruction Duplication

Since the DC-Widths of the 1st dominant DCs are greater than two during 55% of program execution we would need at least one wider cluster to exploit maximum ILP. To address this issue, we propose an instruction duplication algorithm that splits the dominant DCs by duplicating instructions and thereby reducing the DC-Widths of the dominant DCs. This aids us in designing

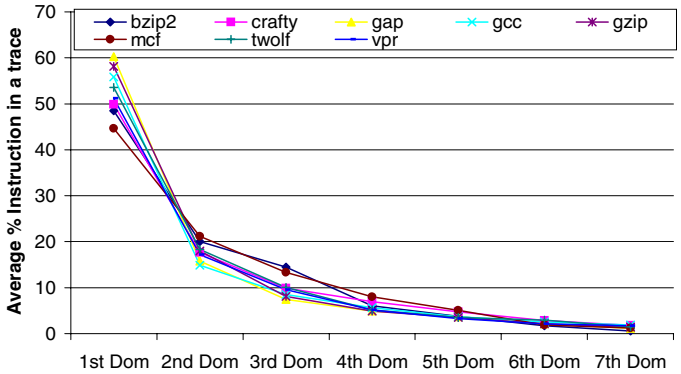

Figure 5: Percentage of instruction distribution in a trace among the top 7 dominant DCs.

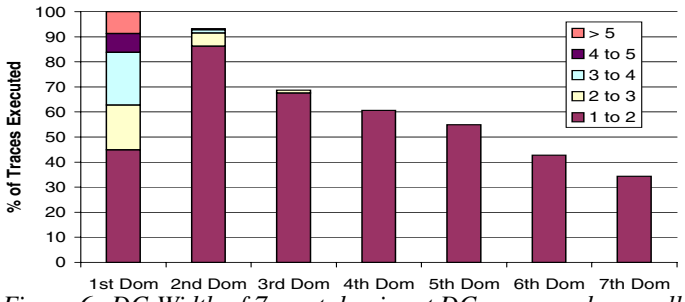

Figure 6: DC-Width of 7 most dominant DCs averaged over all the traces.

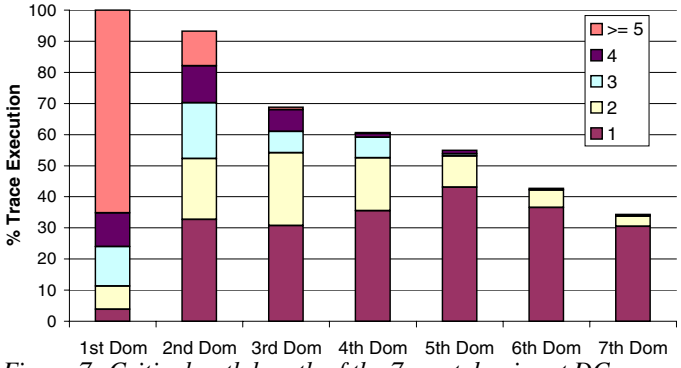

Figure 7: Critical path length of the 7 most dominant DCs averaged over all the traces.

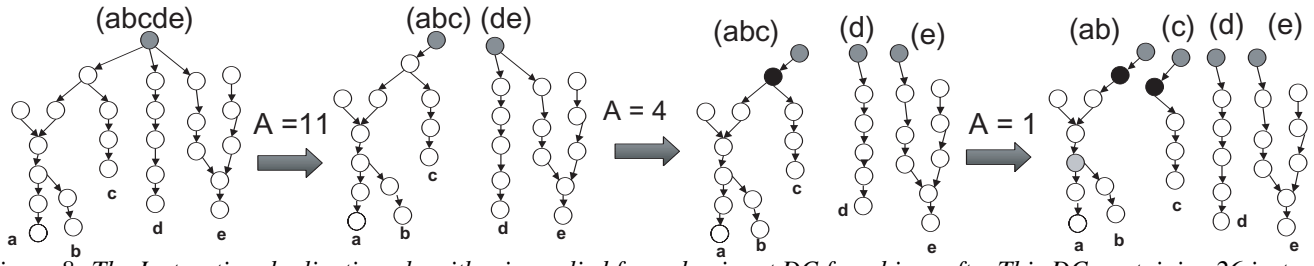

Figure 8: The Instruction duplication algorithm is applied for a dominant DC found in crafty. This DC containing 26 instructions is split into four DCs by duplicating just 4 instructions and the resultant 1st dominant DC has only 10 instructions. "A" stands for the advantage number. Light grey node in the resulting dominant DC (ab) is a potential split point for another split. We do not do this split as the advantage number for doing this split would be negative (6 instructions need to be duplicated to get a reduction of just 2 instructions in the dominant DC).

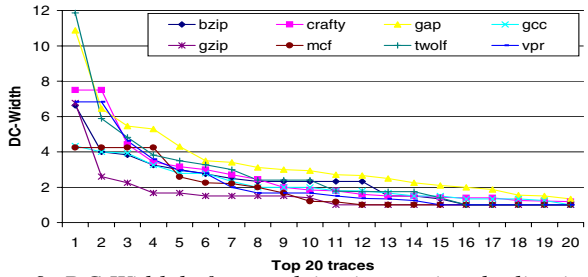

Figure 9: DC-Width before applying instruction duplication optimization. Results are shown only for the 1st most dominant DCs in the top 20 traces. Traces are sorted according to their DC-Width for better readability.

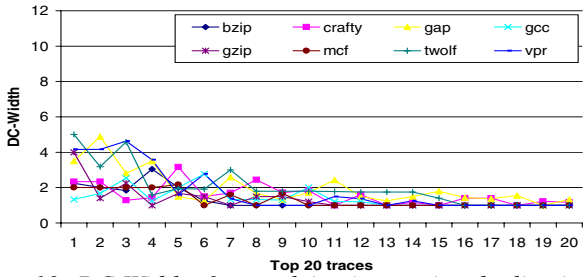

Figure 10: DC-Width after applying instruction duplication optimization.

narrow width clusters without degrading performance.

Our instruction duplication algorithm works as follows. For each leaf in the data dependency graph of the DC that needs to be split, we construct a Leaf-DAG (LDAG). LDAG of a leaf is a subset of instructions in the DC that are necessary to produce input values to execute that leaf instruction. The set of LDAGs, each corresponding to a leaf in the DC, is the maximum possible split for the given DC. For example, let us consider the DC in Figure 8. Let (abcde) represent the original DC where a, b, c, d and e represent the LDAGs of the five leaves. Splitting it into 5 different DCs, (a)(b)(c)(d)(e) is the maximum possible split. This split would result in a large number of duplicated instructions.

If two LDAGs share a large number of instructions, then to reduce the overhead of duplicated instructions, the two should be combined into one DAG. Thus, we need to find the set of LDAGs that needs to be combined into one DAG. The goal is to ensure that the resulting largest DAG among these set of DAGs is as small as possible while at the same time ensuring that the number of duplicated instructions is kept to minimum. In essence, we have converted the problem of splitting DCs into a problem of grouping LDAGs into sets. To find the sets of LDAGs we first start by having all the LDAGs in one set. In the next step, we divide this one set of LDAGs into two sets. To identify the members in each set we make use of a greedy heuristic based algorithm. We compute, what we call, the advantage number for all possible groupings.

The *advantage number* is the number of real instructions in the resulting smaller set minus the number of duplicated instructions. We choose the division that has the highest advantage number. In our example, for the first split (abc)(de), the advantage number, represented as A is 11. We then iteratively apply the same algorithm over the resulting most dominant DCs in each iteration. We terminate the process when the dominant DC could not be split further with a positive advantage number. One optimization we add to this base greedy algorithm is to have a look-ahead of 1. That is, while we calculate the advantage number for the current split, we also calculate the advantage number for the best possible split for the resulting dominant DC and add it up to the current advantage number.

Figure 9 presents the DC-Width of the dominant DC in the top 20 most frequently executed traces sorted according to their DC-Width number. We can see that the DC-Width of the dominant DC goes up to a maximum of 12. After the application of our instruction duplication algorithm, we are able to significantly reduce the DC-Width for such DCs, reducing it to only 2 or less in most cases as shown in Figure 10. Figure 11 summarizes the effectiveness of our duplication algorithm in terms of reducing the number of instructions in the 1st dominant DCs averaged across all traces. The first bar is the average number of total instructions in a trace. The 2nd bar shows the increase in this number as result of additional duplicated instructions introduced. We can see that this additional duplication is less than 10% of the instructions in the traces. The next two bars show the number of instructions in the dominant DC before and after our duplication optimization, respectively. On an average about 20% of the size of the dominant DC has been reduced. The last two bars show the number of instructions in the 2nd dominant DC before and after the application of our algorithm. The number of instructions in the 2nd dominant DC increases because a DC resulting from the split of the original 1st dominant DC may become the new second dominant DC.

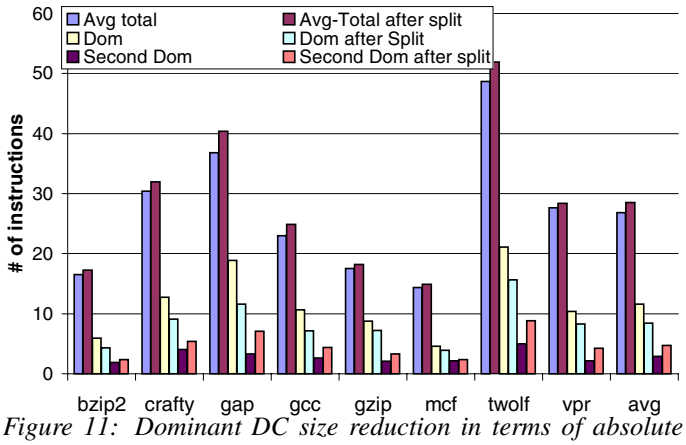

Figure 11: Dominant DC size reduction in terms of absolute number of instructions after the application of instruction duplication algorithm.

7 Dependency Chain Processor (DCP)

The Dependency Chain Processor (DCP) is a clustered microarchitecture design with a trace-based execution model. Each cluster in DCP is a superscalar in-order pipeline. These clusters can be heterogeneous, that is, each of them can be of different issue width. In this section we'll present a model for our DCP microarchitecture and then discuss the representation of the set of DCs that make up a trace.

7.1 Microarchitecture

Figure 12 presents the DCP microarchitecture with 3-wide, 2-wide and 1-wide pipeline clusters.

7.1.1 Fetch

The traces constructed as described in Section 4 may not cover all the program paths as they represent only the hot paths. Therefore, a DCP needs to operate in two modes of execution - DC-Mode to execute hot paths and non-DCMode to execute cold paths. In the non-DCMode, DCP fetches instructions sequentially from the instruction cache and to keep the design complexity efficient, all the instructions are issued in-order to the widest available cluster while the other clusters are left unutilized. When the trace predictor predicts that a trace could be executed the processor switches to DCMode. In this mode, instructions are fetched from the DC-Cache instead of from the I-cache and the DCs are mapped to appropriate clusters based upon a predetermined cluster-ID which will be described in Section 7.2.

7.1.2 Execution Model

The DCs in a trace are mapped to different clusters and they execute independent of each other, as there is no data dependency between them. Though each cluster is an in-order execution pipeline, the DCs can execute at different rates in different clusters out-of-order and thereby achieving higher ILP.

For the baseline DCP microarchitecture we assume serialized execution of traces. That is, we ensure that all the DCs in the currently executing trace complete their executions before DCs from the next trace could be issued. This would mean that we incur a synchronization cost in the form of lost parallelism but the longer traces that we constructed would help mitigating this cost. Serializing the execution of traces ensures that there is no dependency between concurrently executing DCs at any instant of time.

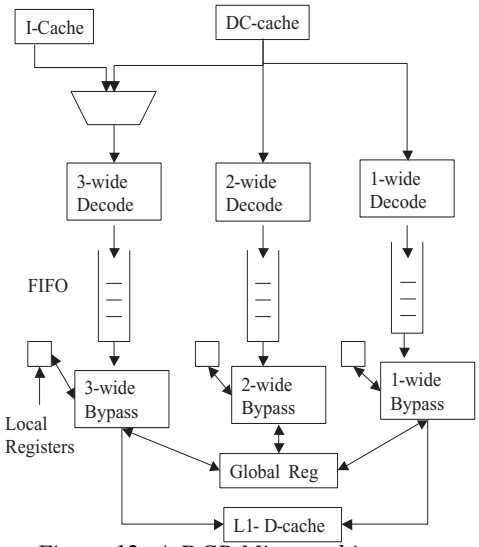

Figure 12: A DCP Microarchitecture

This model of executing DCs in the clusters enables us to aggressively cluster resources from the decode stage to the commit stage unlike in traditional clustered architectures where typically only the functional units are clustered. A heavily clustered architecture would ensure that the complexity of all the pipeline stages are equally scaled down and hence can be clocked at a higher frequency.

7.1.3 Register File Organization and Inter-Cluster Communication

The DCP microarchitecture uses two level hierarchical register files. Each cluster has one 8-entry local register file and there is a global register file accessible by all the clusters.

The global register file is used for reading the live-in and writing the live-out values of the DCs. We assume just 2 read and 2 write ports for the global register file shared by all the clusters and very conservatively assume 3-cycle access latency. We find that the DCP is insensitive to this latency because concurrently executing DCs are totally data independent of each other and would not need to communicate values between them. A live-out value produced by a currently executing DC is going to be read as a live-in only by a DC in the trace to be executed in the future.

Intermediate values produced while executing a DC in a cluster are written and read from the local registers of the cluster. For a 3-wide cluster, the local register file is required to have 6 read and 3 write ports to issue 3 instructions in one clock cycle. By capturing the intermediate values in local registers, we reduce the bandwidth to the global register file that is accessed by all the clusters.

Since our machine model is not sensitive to the latency of the global register file, one could possibly design clustered microarchitectures without a global register file by communicating live-in and live-out values of DCs through memory using additional load/store instructions.

	6-way In-order superscalar	DCP	ILDP	6-way OOO superscalar
Decode bandwidth	6-wide	Maximum 3-wide	6-wide	6-wide
Rename bandwidth	NONE	NONE	6 read/write ports to Map table	18 read/write port to map table
Steering Logic	NONE	NONE	Accumulator based	Complex dependence-based (if clustered)
Issue Logic	In-order	In-order	In-order	6-way out-of-order, Issue Q: Integer: 32 FP: 32
Register File	128 entry with 6 write and 12 read ports	128 entry global register file with 2 read/ 2 write ports. (For 3-wide cluster: 3 write and 6 read ports) 8-entry local register file per cluster to hold intermediate values.	128 entry global register file with 1 read/ 2 write ports, and is replicated 6 times, one per cluster. One accumulator per cluster	128 entry with 6 write and 12 read ports
Bypasses	Equivalent to 6-wide machine	Equivalent to 3-wide machine	Equivalent to 1-wide machine	Equivalent to 6-wide machine
ROB	NONE	NONE	128 entries	128 entries
Fetch	I-Cache, Trace Cache,Multiple branch predictor	I-cache, DC-cache, Multiple branch predictor	I-cache, Branch predictor	I-cache,Branch predictor

Table 1: Complexity of In-order superscalar, DCP, ILDP and OOO-superscalar processors

7.1.4 Commit and Recovery from Mis-speculation

Unlike ILDP [11] and out-of-order processors, we do not require a re-order buffer. We do not need it in the non-DCMode as instructions execute in program order. In DCMode, instructions are committed only at the trace boundaries, which is possible using a shadow copy for the global register file. If there is an exception or a misprediction while executing a trace, we switch to non-DCMode and start re-executing instructions from the beginning of the last committed trace using register values in the shadow register file. The dependency chain to which a load/store should belong to is determined through profile analysis. A memory order checker is used to verify memory dependences and recover at trace boundaries in case of assumed memory order violation.

7.1.5 Complexity Effectiveness

Table 1 compares the complexity effectiveness of our microarchitecture versus previously proposed ILDP microarchitecture [11] and traditional clustered out-of-order and in-order microarchitectures.

Palacharla et al., [18] noted that rename, issue and bypass logic are the most performance critical structures in the pipeline. The DCP does not have the complexity of renaming and steering/issue logic as these functions are performed during offline construction of DCs.

Complexity of bypass logic increases quadratically with respect to increase in issue widths [18]. Over our baseline 6-wide in-order design, the DCP has less complex bypass logic as it uses at most 3-wide clusters and there is no requirement for any direct bypasses between clusters. Also, a re-order buffer is a not required unlike ILDP and out-of-order processors for the reasons explained in previous subsection.

Balasubramonian et al., [5] observed the need to reduce the complexity of the register file in wide-issue processors. Current IPF implementations with a 6-wide pipeline have a 128-entry register file with 12 read and 6 write ports to issue 6-instructions in a clock cycle. Whereas, in DCP each cluster has a small 8-entry local register file with at most 6 read and 3 write ports (for a 3-wide cluster). Additionally, DCP has a 128-entry global register file with just 2 read and 2 write ports as local register files handle most of the traffic. We analytically quantify the area and energy advantages of such a register file organization.

We can calculate the relative area used by register files by using the formula: $entries * (WP + RP)^2$, where, $entries$ is the number of entries in the register, WP is the number of write ports and RP is the number of read ports. This formula concurs with published results in academia [22] that found the register file size grows with the square of the number of ports especially for register files with many ports. Using this formula, we found the optimal design point for implementing a 128-entry register file with 12 read and 6 write ports. This results in an implementation with two replicated 128-entry register files each with 6 read and 6 write ports that would together provide 12 read and 6 write ports. To implement an 8-entry local register file with 6 read and 3 write ports in DCP, an efficient implementation would have two replicated 8-entry register files each with 3 read and 3 write ports. We find that register file in DCP architecture with two 3-wide clusters, will occupy 11.5 times less area than the register file in a 6-wide processor and about 2 times less area than the one in the ILDP architecture.

Using a similar construction we can compute the relative energy requirements of each register file configuration. Register file access energy is proportional to CV^2, where C is the capacitive load of the access logic and V is the supply voltage. Given a fixed supply voltage and a wire-congested register file design, relative energy per port per access will change in proportion to wordline and bitline lengths, which are proportional to $sqrt(entries) * (WP + RP)$. The maximum energy requirements of each register file, assuming all ports on all copies accessed, is proportional to $sqrt(entries) * (WP + RP)^2$. Using this formula we find that energy used in DCP due to register file could be up to 5.5 times less when compared to the register file in a 6-wide processor.

7.2 DC-Trace Representation

We assume that DC-Traces are constructed offline. Each DC-Trace consists of a set of Dependency Chains (DCs), which need to be represented in a format that can be efficiently consumed by the DCP microarchitecture. Such a method to represent DCs could be viewed as an additional means for the compiler to com-

municate parallelism to the in-order clustered microarchitecture.

Each DC in the trace needs to be mapped to a particular cluster in the microarchitecture. If there are more DCs than available clusters then more than one DC will be mapped to a single cluster. Since DCP can have heterogeneous clusters we need a judicious and a simple mapping policy. We use the following greedy algorithm. Let W1, W2, W3, W4 represent the configuration of the cluster microarchitecture where Wi represents the issue width of the i-th cluster. DCs are sorted according to their dominance and mapped one by one onto the clusters in that order. Let us say, after mapping some DCs, N1, N2, N3, N4 represents the total number of instructions mapped to corresponding clusters. While mapping a DC to a cluster, we calculate cluster-load as Ni/Wi and map the DC to that cluster that has the least cluster-load value. If two clusters have the same cluster-load value, then we choose the one with greater width. For example, while trying to map the 1st dominant DC, all Ni will be zero. Hence we'll map the most dominant DC to the widest available cluster.

After mapping a DC to a cluster, the DC is appended with a cluster-ID that uniquely identifies the cluster to which it is mapped. While executing a trace, a particular DC in the trace can be steered towards a particular cluster based on this cluster-ID. This cluster scheduling policy in hardware is much simpler when compared to other dynamic hardware-based scheduling policies [6].

To schedule instructions within a DC, we do a simple level scheduling based on its dependency graph. Load/store dependences are resolved based on their first instances of execution while collecting instruction profiles. We find this simple disambiguation to be effective for most of the cases. But if this assumption is violated during program execution, recovery is done at the trace boundary as explained in Section 7.1.4.

It is also necessary to map the instructions in the trace to the corresponding register file (local or global). Intermediate values produced by DCs are written and read from the local register file which is shared between multiple DCs executing in the same cluster. Live-in (values produced by DCs in previously executed traces) and live-out are referred to the global register file. Values produced by duplicated instructions are always renamed to the local register file as their values are used only within a DC and will not modify other architectural states.

8 Performance Evaluation

In this section we will analyze the performance potential of our DCP microarchitecture.

8.1 Baseline Configuration

We model processor performance using cycle accurate SMTSIM/IPFSim, a modified version of the SMTSIM [21] simulator that has been enhanced to work with Itanium binaries.

All the benchmarks are simulated for 100 million retired instructions after fast-forwarding for 1 billion instructions. All the memory requests during fast-forwarding are tracked and simulated in order to warm up the caches past initialization.

Table 2 presents our simulator configurations. The data caches are multi-way banked and all the caches are non-blocking

		Stages	Misprediction Penalty	Mis-fetch Penalty
Pipeline Structure	In-order	10	8	2
	OOO	14	12	2
	DCP	10	8	2
Fetch Width	6 inst			
Memory	230 cycles; TLB Miss penalty 30 cycles			
Branch Prediction	16K entry PHT, 12-bit ghr GSHARE 1024 entry 4-way associative BTB			
Register Files	128 Int; 128 FP			
Cache Hierarchy	L1 D-cache: 16K 4-way, 8 way banked, 1 cycle L1 I-Cache: Ideal L2 (unified): 256K 4-way, 8 way banked, 14 cycles L3 (unified): 3072K 12-way, 1 way banked, 30 cycles All caches have 64 byte lines. Data caches are write-back and write-allocate.			

Table 2: Simulator Configuration

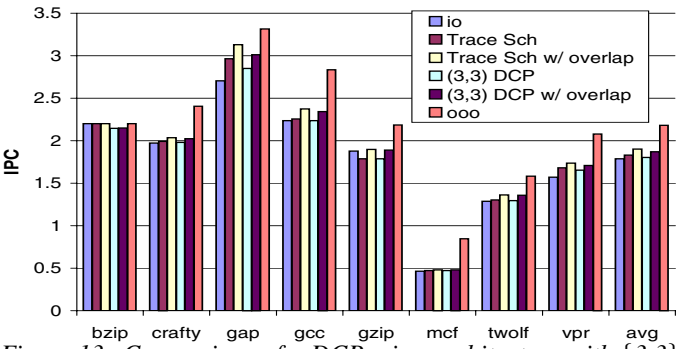

Figure 13: Comparison of a DCP microarchitecture with {3,3} clusters to the baseline configurations

with up to 16 misses in flight at once. For out-of-order processor models, we assume two issue queues one for floating point instructions and the other for integer instructions each of size 32.

Trace based DCP microarchitecture requires a trace predictor to predict the traces that can be executed in future. In our simulations, we make trace predictions by predicting the outcomes of the required number of future branches by serially accessing the branch predictor. If any one of these branches were mispredicted, we discard all the useful instructions executed as part of the trace and perform recovery at the trace boundaries. In the DCMode of the DCP microarchitecture, instructions need to be fetched from the DC-Cache. In order to factor out the advantages of improved I-cache performance in the DCP microarchitecture we assumed perfect I-cache in both baseline and DCP microarchitecture simulations. A similar approach was adopted by Kim and Smith [11] in their studies.

8.2 Performance Analysis of DCP microarchitecture

Figure 13 compares the performance of the DCP microarchitecture with a {3,3} configuration ({3,3} represents two 3-wide in-order superscalar pipelines) with in-order and out-of-order superscalar processors. *Trace-Sch* in Figure 13 is the baseline configuration, which is an in-order 6-wide processor that can execute optimized trace schedules. It should be noted that this baseline configuration would also require fetch support as discussed in Section 7.1.1. The duplicated instructions that get executed in DCP microarchitecture are not counted (in terms of instructions) when calculating IPC, but they are modeled correctly and effect the number of cycles it takes to execute the program.

Executing scheduled traces results in up to 15% improvement (for gap) in IPC for a 6-wide in-order processor. This could be noted by comparing *Trace-Sch* with *io* as shown in Figure 13. From the Figure 13, it is also clear that the DCP microarchitecture with two 3-wide in-order clusters achieves an IPC equivalent to that of a micro-architecture using one 6-wide cluster with trace scheduling (represented as *Trace-Sch*). This is made possible by executing optimized DCs ensuring more uniform distribution of instructions among clusters. Further, unlike in a 6-wide in-order cluster where one instruction could stall the entire pipeline, DCs in two 3-wide in-order pipelines could execute independent of each other - that is, even if execution of a DC is stalled in one cluster, the other clusters could continue executing the DCs assigned to it. In addition, as was discussed earlier in Section 7.1.5, the {3,3} DCP microarchitecture has several complexity advantages over the 6-wide monolithic microarchitecture optimized for trace scheduling.

All the experiments in Figure 13 assume 3-cycle latency for accessing the global register file. As explained in Section 7.1.3, our microarchitecture is insensitive to this latency since the DCs of a trace executing simultaneously are data independent of each other. A value produced by a DC in a cluster will not be required by other DCs executing in parallel in other clusters. It will be required in other clusters only while executing the DCs from the next DC-Trace. We found that assuming 0-cycle latency for communication provides negligible IPC improvements (not reported here due to space constraints).

8.2.1 Overlapping DC-Traces

As described in Section 7.1.2 we had assumed serialized execution of traces for the baseline DCP model. We also study the performance improvement that could be attained if the executions of the traces are overlapped. That is, while executing DCs from the current trace, DCs from the next trace can also be allowed to execute. This would mean that there could be data dependences between concurrently executing DCs in different clusters which would complicate the hardware design. But in order to analyze the performance potential of such an execution model, we experimented by assuming 3-cycle communication delay between the clusters. The result for this experiment is presented as *DCP w/ overlap* in Figure 13 which shows 10-15% improvement in IPC over baseline DCP model. This improvement is possible only at the cost of increased complexity in hardware. That is, if a DC in one cluster is data dependent on a DC executing in another cluster, then we would need a pipeline interlock mechanism to detect the dependency and stall the execution of the dependent DC. When the producer instruction finishes its execution the data needs to be bypassed across the cluster and a wakeup logic will also be required to restart the execution of the stalled DC.

8.3 Heterogeneous Clustered Microarchitectures

Figure 14 shows IPC results comparing {3,3} with {3,2,1} and {2,2,2} configurations. Performance of a DCP microarchitecture with {3,2,1} configuration is found to be almost equivalent to that of a {3,3} configuration which is consistent with our characterization of DCs. The DC analysis in Section 5, showed that there is usually only one dominant DC in a trace that has its DC-Width greater than 2. So, just one 3-wide cluster is sufficient to exploit most of the available ILP.

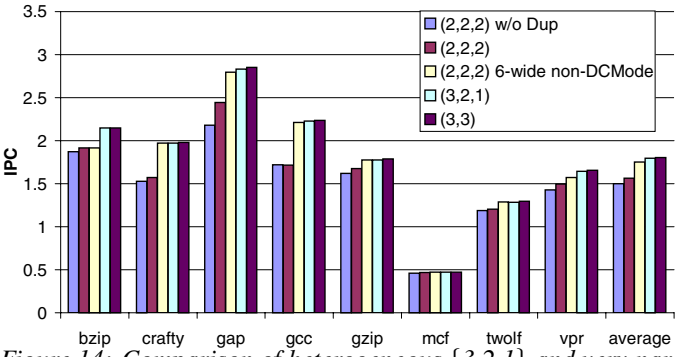

Figure 14: Comparison of heterogeneous {3,2,1} and very narrow width {2,2,2} configurations to {3,3} DCP configuration.

There are several advantages in a heterogeneous cluster microarchitecture. A {3,2,1} configuration will require less area than a {3,3} configuration as one can design {3,2,1} with a smaller bypass network logic. As a result, {3,2,1} will also consume less energy and also opens up more opportunities to reduce power through clock gating the unused clusters when there is only limited ILP available.

Though heterogeneous clusters have the above advantages, homogeneous configurations like {3,3} are still easier to design than a heterogeneous {3,2,1} configuration because to implement the former one needs to just replicate the design for a 3-wide cluster. But to implement the latter one might need to design essentially 3 different pipelines for three different widths. Hence this presents an interesting tradeoff between the required design effort versus saving area/energy.

8.4 Narrow Width Clusters and Efficiency of Instruction Duplication Algorithm

Figure 14 also compares the {3,3} configuration with {2,2,2}. We can see that there is significant reduction in performance in the {2,2,2} configuration when compared to {3,3}. This is primarily due to the fact that in the non-DCMode all the instructions are executed in the widest available cluster, which in the case of {2,2,2} is 2-wide. A better trace formation algorithm with a higher execution coverage would solve this problem. The result labeled "(2,2,2) 6-wide non-DCMode" presents the result for {2,2,2} configuration but assumes 6-wide for non-DCMode. This result is just to elucidate the fact that there is no significant performance degradation in DCMode for {2,2,2}, which implies that DC construction is efficient for {2,2,2} configuration as well. The instruction duplication algorithm discussed in Section 6, resulted in DCs with DC-width mostly under 2.

We use Figure 14 to also bring out the performance advantage resulting from using instruction duplication algorithm. The first bar in the graph is for {2,2,2} configuration simulated without applying instruction duplication optimization. It shows that instruction duplication algorithm provides performance advantage up to 13% (for gap) about 5% on an average.

Finally, Figure 15 shows the percentage of dynamic instructions that are executed in the DCMode, which is common for all the configurations. It also shows the percentage of dynamic

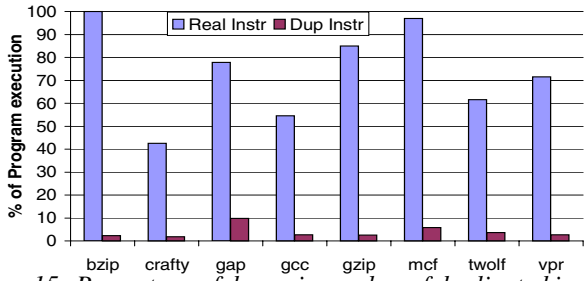

Figure 15: Percentage of dynamic number of duplicated instructions compared with the original number of instructions issued in DCMode.

number of duplicated instructions, which is about 3% on average. This implies that the overhead incurred due to duplicated instructions is very minimal.

9 Conclusions

In this paper, we propose a Dependency Chain Processor (DCP) to explore clustered microarchitecture designs for future EPIC architectures. A key feature of this design is that pipeline resources are clustered from the decode to the commit stage without requiring any direct bypasses between them. This is made possible by assuming trace based execution model where coarse-grained mutually data independent dependency chains are executed in multiple clusters. We envision that optimized DCs could be an additional means for the EPIC compiler to expose parallelism to a clustered microarchitecture.

We show that DCP microarchitecture has the potential to improve both IPC and frequency relative to the current monolithic 6-wide in-order designs. IPC is improved (up-to 15%) through scheduling of instructions within a trace while frequency improvement is achieved by implementing fairly narrow (up to 3-wide) in-order pipelines that do not require inter-cluster bypass hardware.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded by Intel and by NSF grant No. CCR-0311712.

References

[1] Aneesh Aggarwal and Manoj Franklin. Instruction replication: Reducing delays due to inter-pe communication latency. In *IEEE PACT*, pages 46–55, 2003.

[2] Alex Aleta, Josep M. Codina, Antonio Gonzalez, and David Kaeli. Instruction replication for clustered microarchitectures. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 326. IEEE Computer Society, 2003.

[3] Alex Aleta, Josep M. Codina, Jesus Sanchez, and Antonio Gonzalez. Graph-partitioning based instruction scheduling for clustered processors. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 150–159. IEEE Computer Society, 2001.

[4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12. ACM Press, 2000.

[5] Rajeev Balasubramonian, Sandhya Dwarkadas, and David H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 237–248. IEEE Computer Society, 2001.

[6] Rajeev Balasubramonian, Sandhya Dwarkadas, and David H. Albonesi. Dynamically managing the communication-parallelism trade-off in future clustered processors. In *ISCA*, pages 275–286, 2003.

[7] Nevill-Manning C.G. Inferring sequential structure. *Phd dissertation. University of Waikato, NZ*, 1996.

[8] Jamison D. Collins and Dean M. Tullsen. Clustered multithreaded architectures . pursuing both ipc and cycle time. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[9] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic. The multicluster architecture: reducing cycle time through partitioning. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 149–159. IEEE Computer Society, 1997.

[10] Daniel Holmes Friendly, Sanjay Jeram Patel, and Yale N. Patt. Putting the fill unit to work: dynamic optimizations for trace cache microprocessors. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 173–181. IEEE Computer Society Press, 1998.

[11] Ho-Seop Kim and James E. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *Proceedings of the 29th Annual International symposium on Computer architecture*, pages 71–81. IEEE Computer Society, 2002.

[12] Ho-Seop Kim and James E. Smith. Dynamic binary translation for accumulator-oriented architectures. In *Proceedings of the international symposium on Code generation and optimization*, pages 25–35. IEEE Computer Society, 2003.

[13] Ilhyun Kim and Mikko H. Lipasti. Macro-op scheduling: Relaxing scheduling loop constraints. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 277. IEEE Computer Society, 2003.

[14] James R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 259–269. ACM Press, 1999.

[15] Chi-Keung Luk, Robert Muth, Harish Patil, Robert S. Cohn, and P. Geoffrey Lowney. Ispike: A post-link optimizer for the intel itanium architecture. In *In the Proceedings of International Symposium on Code Generation and Optimization*, pages 15–26, 2004.

[16] Cameron McNairy and Don Soltis. Itanium 2 processor microarchitecture. *IEEE Micro*, 23(02):44–55, 2003.

[17] R. Nair and M. Hopkins. Exploiting instruction level parallelism in processors by caching scheduled groups. In *24th Annual International Symposium on Computer Architecture*, pages 13–25, June 1997.

[18] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th annual international symposium on Computer architecture*, pages 206–218. ACM Press, 1997.

[19] Steven E. Raasch, Nathan L. Binkert, and Steven K. Reinhardt. A scalable instruction queue design using dependence chains. In *Proceedings of the 29th Annual International symposium on Computer architecture*, pages 318–329. IEEE Computer Society, 2002.

[20] Michael S. Schlansker and B. Ramakrishna Rau. Epic: Explicitly parallel instruction computing. *Computer*, 33(2):37–45, 2000.

[21] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. *In the Proceedings of the 22th Annual International Symposium on Computer Architecture*, 23(2):392–403, 1995.

[22] Victor V. Zyuban and Peter M. Kogge. Inherently lower-power high-performance superscalar architectures. *IEEE Transactions on Computer*, 50(3):268–285, 2001.