Cluster-Level Simultaneous Multithreading for VLIW Processors

Manoj Gupta, Josep Llosa, Fermín Sánchez[1]

* Computer Architecture Department, UPC, Barcelona, Spain

ABSTRACT

Clustered VLIW embedded processors have become widespread due to benefits of simple hardware and low power. However, while some applications exhibit large amounts of instruction level parallelism (ILP) and benefit from very wide machines, others have little ILP, which wastes precious resources in wider processors. Simultaneous MultiThreading (SMT) is a well known technique that improves resource utilization by exploiting thread level parallelism at the instruction grain level. However, implementing SMT for VLIWs requires complex structures. In this paper we propose CSMT (Cluster-level Simultaneous MultiThreading) to allow some degree of SMT in clustered VLIW processors with minimal hardware cost and complexity. Our approach uses the "bundle" (the set of operations that execute simultaneously in a given cluster) as the minimum assignment unit. Additionally, all bundles belonging to a VLIW instruction execute simultaneously. To minimize cluster conflicts between threads, a very simple hardware-based cluster renaming mechanism is proposed. The experimental results show that CSMT significantly improves ILP when compared with other multithreading approaches suited for VLIW. For instance, with 4 threads CSMT shows an average speedup of 118% over a single threaded VLIW and 41% over Interleaved MultiThreading(IMT). In some cases, speedup can be as high as 195% over single thread and 88% over IMT.

KEYWORDS: Clustering, VLIW processor, Multithreading

1 Introduction

Very Long Instruction Word (VLIW) is a paradigm for exploiting Instruction Level Parallelism (ILP) based on exposing the architecture details to the compiler, so that ILP can be extracted at compile time. Therefore, contrary to superscalars, no special hardware like register renaming, instruction queues, reorder buffers, etc. is required. VLIWs have been used in general purpose computing [6, 19, 12]. However, due to the hardware simplicity, low cost and low power consumption, the VLIW paradigm has found its niche in embedded computing [8, 17, 7].

[1]E-mail: {mgupta,josepll,fermin}@ac.upc.edu

Many embedded applications exhibit significant amounts of ILP, or at least regions with high ILP interleaved with low ILP regions. Therefore, to exploit such ILP, VLIWs need to be designed with a significantly wide issue width, which is limited by the number of functional units (FUs). However, the number of FUs is limited by the scalability of the register file and the complexity of the bypassing network. Register file access time grows linearly with the number of ports, while area grows quadratically with the number of ports, which are proportional to the number of FUs. Bypassing network can impact area and processor cycle time in a similar way. Clustered VLIW architectures tackle this problem by introducing more than one register file and clustering the FUs according to the register files they are connected to. This approach allows higher levels of issue width than unicluster VLIW architectures, since register file ports and bypass network are determined by cluster width. While cluster width can be kept low, issue width can be easily scaled by increasing the number of clusters. Many VLIWs have been designed from scratch using the clustered approach [8, 9, 15].

However, the ILP exposed in many other applications, or in some code regions, is limited and the processor is heavily under utilized. Simultaneous MultiThreading (SMT) [22] is a well known technique to improve the resource utilization by exploiting thread level parallelism (TLP). Although SMT at the operation level has been proposed for VLIWs [13], it requires some sort of out of order execution [18], which significantly increases processor complexity, taking away most of the advantages of VLIWs. In order to maintain the VLIW simplicity, simpler multithreading techniques have been proposed.

Block multithreading [16, 2, 21] executes VLIW instructions from a single thread until it is blocked by an event (a cache miss, for e.g.). When that happens, a fast context switch gives control to a different thread so that most of the miss latency is hidden. However, there are still a few vertical slots wasted due to context switch time. In addition, no horizontal empty slot inside a VLIW can be used for other threads.

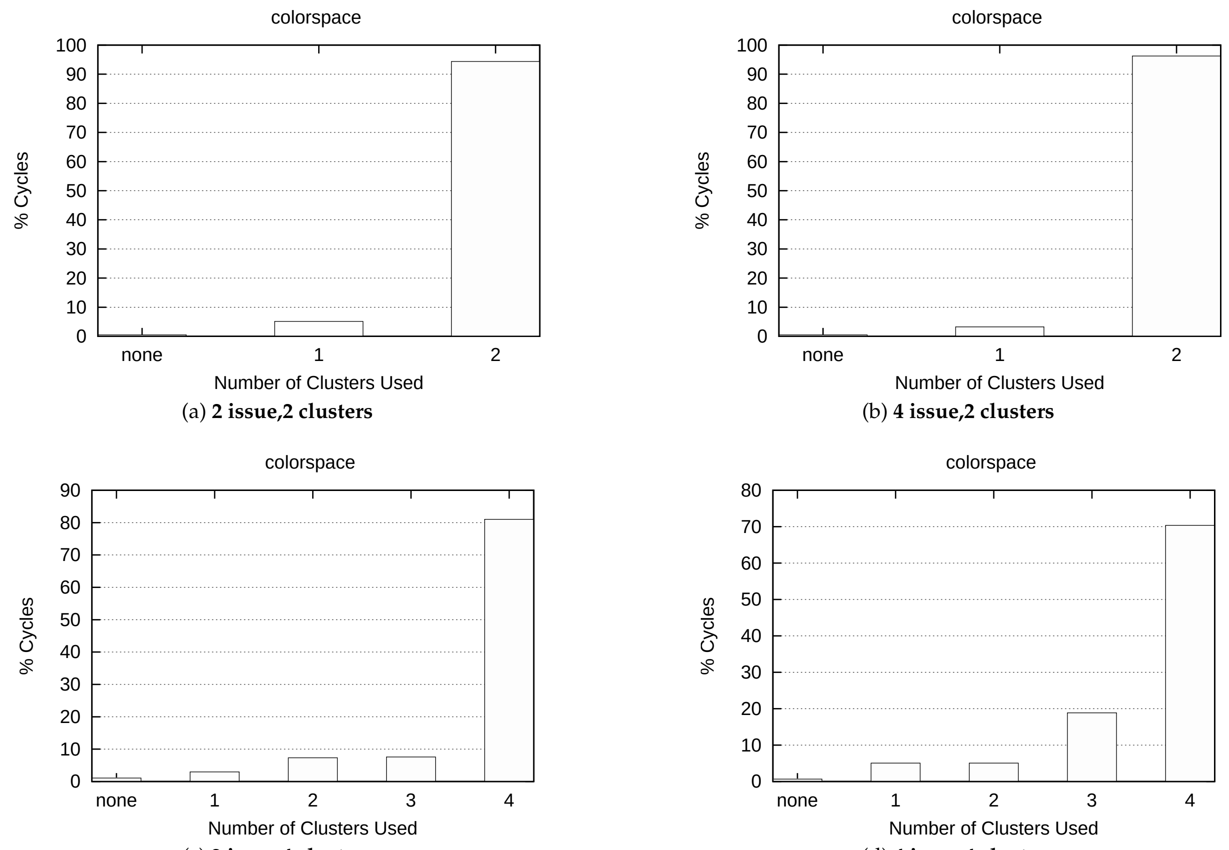
Interleaved multithreading [20, 3] does a zero cycle context switch every cycle, so that VLIW instructions from different threads are "interleaved" at execution time. Interleaved multithreading allows the removal of the bypass network. However, doing so hinders single thread performance when only one thread is present. In order to hide cache misses, many threads are required. In addition, it still does nothing to remove horizontal waste. A trivial modification consists of marking as blocked any thread that produced a cache miss and issuing VLIW instructions only from non-blocked threads. This modification achieves the best from both block and interleaved multithreading.

In [4], a quite different approach for clustered VLIWs is taken. The processor has two modes: single threaded and multithreaded. In single threaded mode, VLIW instructions are issued from a single thread that has been compiled to make use of all the clusters. In multithreaded mode, short VLIW instructions are issued from multiple threads by executing each thread in a different cluster. In the latter mode, threads have been compiled to use a single cluster. In addition, switching between modes is also compiler (or programmer) controlled. This approach does nothing to avoid vertical waste due to cache misses and cannot exploit TLP between different applications.

Our approach, named CSMT (Cluster-level Simultaneous MultiThreading), tries to exploit TLP at the cluster level without requiring any compiler support. Since it is transparent to the compiler, it can be used either with compiler generated threads or with threads belonging to different applications. CSMT issues simultaneously several VLIW instructions provided that they do not conflict in the clusters they require to execute. In order to reduce cluster conflicts when several threads require the same logical cluster, a very simple hardware-based static cluster renaming technique is used to map logical clusters, belonging to different threads, to different physical clusters. With such approach, a significant amount of horizontal waste can be removed. In addition, by marking threads that produce events like cache misses as blocked, vertical waste is also removed. Finally, CSMT does not degrade single thread performance since all resources are available when a single thread is present.

Rest of the paper is organized as follows. Section 2 presents some statistics that motivates CSMT, as well as a simple example. The base architecture over which we build CSMT is explained in section 3. The details of the proposed architecture are discussed in Section 4. An evaluation and comparison with other approaches is performed in Section 5. Finally, Section 6 concludes the paper.

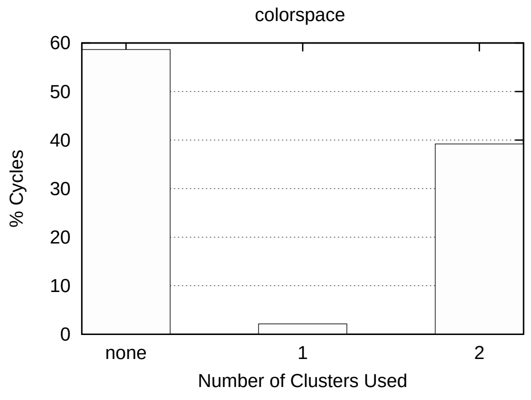
2 Motivation



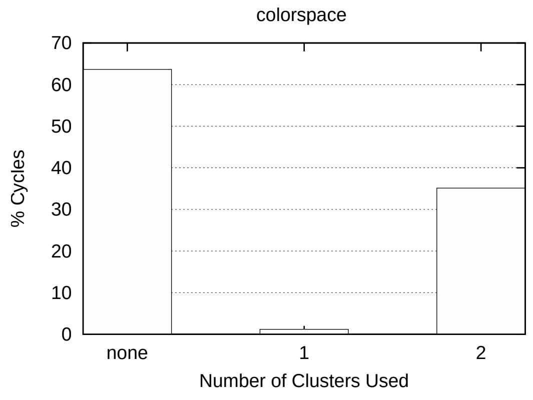
(a) **2 issue,2 clusters** (b) **4 issue,2 clusters** (c) **2 issue,4 clusters** (d) **4 issue,4 clusters**

Figure 1: **Cluster Usage Distribution for colorspace benchmark with perfect memory**

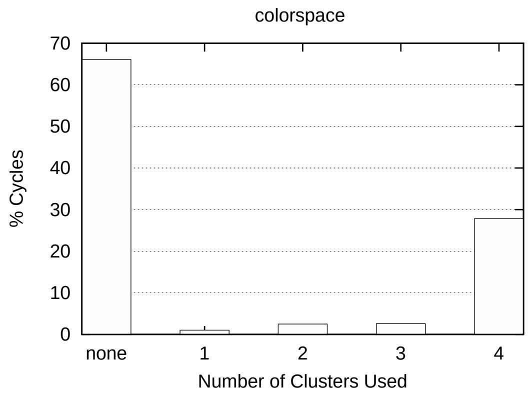
Figure 1 shows the resource utilization based on cluster usage for the benchmark colorspace [1] assuming



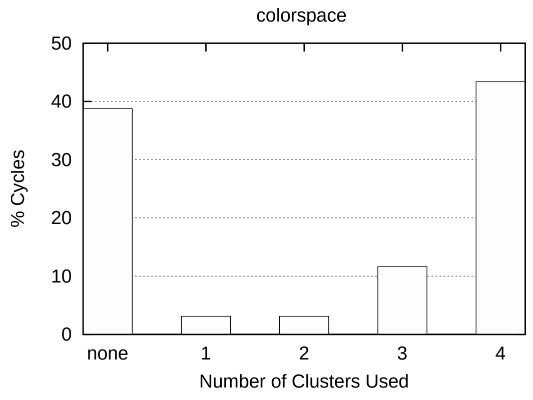
(a) **2 issue,2 clusters**



(b) **4 issue,2 clusters**



(c) **2 issue,4 clusters**



(d) **4 issue,4 clusters**

Figure 2: **Cluster Usage Distribution for colorspace benchmark with real memory**

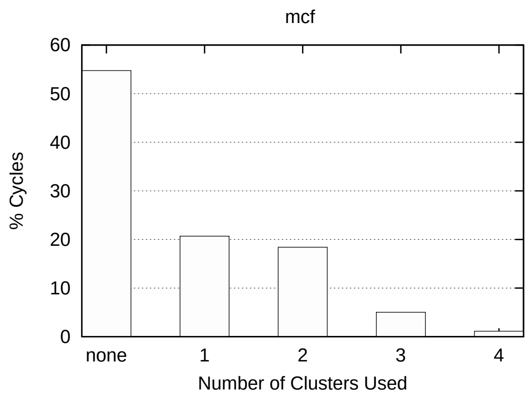
ideal memory. Figures 1(a) and (b) show the percentage of time a given number of clusters is used in a 2-cluster architecture with 2-issue and 4-issue width per cluster respectively; figures 1(c) and (d) show the same data for a 4-cluster architecture. Thus, in figures 1(a) and (b) the label 1 represents the percentage of usage of only one cluster (cluster 0 or cluster 1), while the label 2 represents the time both clusters are simultaneously used. Figures 1 (c) and (d) represent similar data for a 4-cluster architecture. We assume a cluster is used when any of its FUs is used.

Colorspace benchmark has a high ILP degree, close to 7 with a 8-issue width and close to 9 with a 16-issue width. Due to this high ILP, all the clusters are simultaneously used most of the time, as shown in Figure 1.

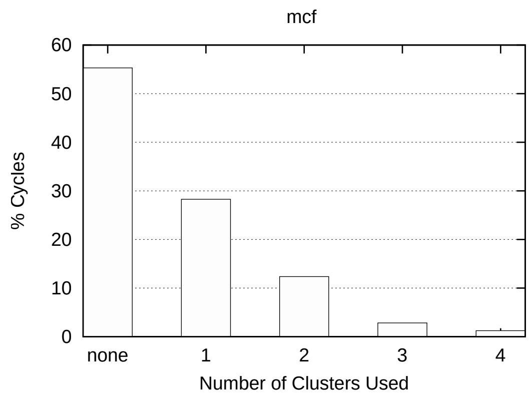
When a real memory model is considered, a significant amount of time is wasted in handling the cache misses (up to 60% of the execution), as can be seen in Figure 2. We have considered a 20-cycle delay for a cache miss latency in this simulation. In order to reduce the time the processor is idle, a simple multithreading technique, like interleaved multithreading [20, 3], can be used to tolerate the cache misses by scheduling other threads during the miss intervals, increasing in that way the processor throughput.

However, other applications exibit in general much lower ILP. The ILP in SPEC [11] or Mediabench [14] benchmarks ranges, in general, between 1 and 2. Thus, when repeating the experiment by using SPEC and Mediabench benchmarks we obtain significantly different results.

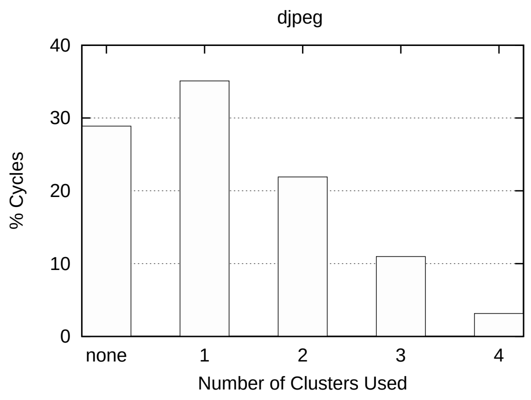
Figure 3 shows the cluster usage in different architectures for two of these benchmarks , the 181.mcf from



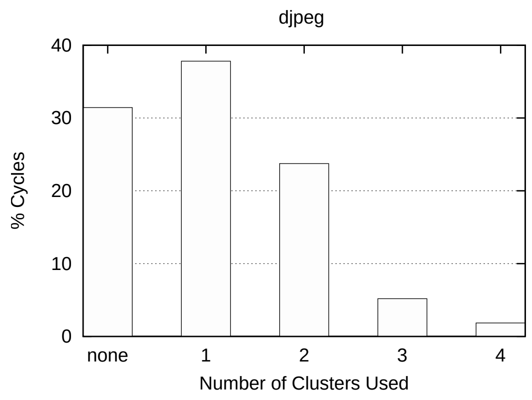
(a) **2 issue,4 clusters**



(b) **4 issue,4 clusters**



(c) **2 issue,4 clusters**



(d) **4 issue,4 clusters**

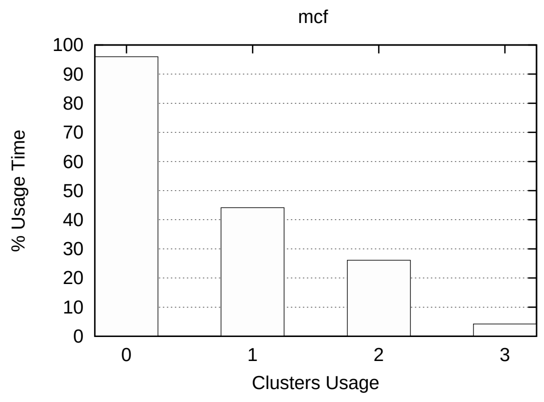
Figure 3: **Cluster Usage Distribution for Jpeg Decoder and Mcf**

Spec2000 (figures 3(a) and (b)) and the Jpeg Decoder in MediaBench (figures 3(c) and (d)). As shown in Figure 3, though a lot of time is still wasted in handling cache misses, the cluster usage is very unbalanced and most of the time a unique cluster is used. This bias on the use of a single cluster increases further with the issue width per cluster (20% to 30% in mcf and 35% to 38% in djpeg). This is reasonable, since the compiler tries to schedule as many operations as possible in a single cluster to avoid communication overhead among different clusters. Only a small number of clusters thus get used most of the time, since there is not always enough ILP available during a program's execution.

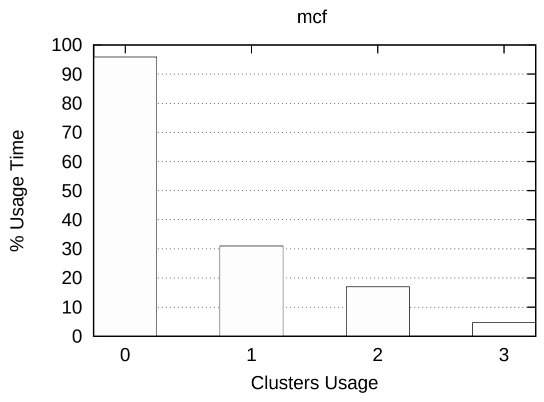
Unlike Figures 2 and 3, Figure 4 represents the percentage of usage of each cluster for 181.mcf and djpeg benchmarks. In other words, the label 2 represents the usage of cluster 2. The figure shows that cluster 0 is the most heavily used and there is little use of other clusters. As can be seen, there is a heavy load imbalance in the programs most of the time.

The use of SMT may improve the cluster utilization but, if we implement SMT in a naive way, most of the threads will compete among resources on a few clusters most of the time, rather than using resources in other clusters which are heavily under-utilized. In fact, most of the time all the threads will compete for the use of cluster 0, as can be derived from Figure 4.

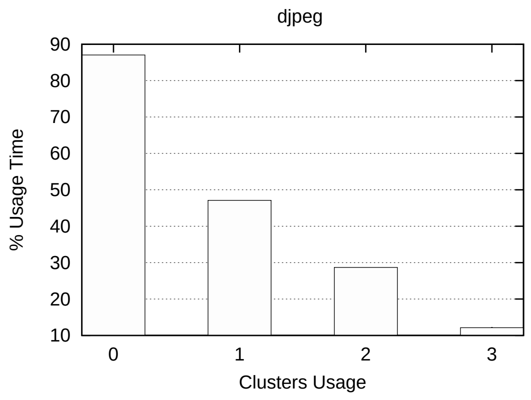
CSMT, the SMT approach proposed in this paper, is based on renaming at execution time the clusters initially assigned by the compiler to each operation. Conflicts are resolved at cluster level instead of FU level,



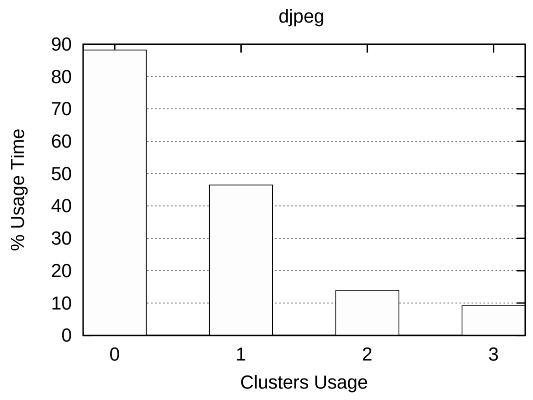
(a) **2 issue,4 clusters**



(b) **4 issue,4 clusters**



(c) **2 issue,4 clusters**



(d) **4 issue,4 clusters**

Figure 4: **Individual Cluster Usage for McF and Jpeg Decoder**

which is significantly easier and cheaper than a fully blown SMT. For example, if two treads (T0 and T1) are using only cluster 0 in a given cycle, operations from cluster 0 in T1 can be assigned to cluster 1, avoiding the conflict. This technique performs the same function as instruction steering logic used in clustered superscalar processors [5], except that our static proposal is much simpler.

Figure 5 shows a sample multithreaded execution for 4 threads on a 4-clustered architecture by using CSMT. Figure 5(a) represent 4 different threads and the cluster assignment (logical clusters) done by the compiler for each bundle (group of operations belonging to a cluster). Letters A-D represent a bundle scheduled in logical cluster 0-3 and the subscript indicates the execution cycle in a single threaded environment.

Figure 5(b) shows the physical mapping of clusters done by the cluster renaming logic for each thread. The logical and physical assignment of clusters is the same for thread 0, but is different for the remainder threads. For example, for Thread 1 (T1), logical cluster 0 (LC0) is assigned to physical cluster 1 (PC1), LC1 to PC2, LC2 to PC3 and LC3 to PC0. In the same way, LC0 from T2 is assigned to PC2 and so on, and LC0 from T3 is assigned to PC3 and so on.

Let us assume that thread 3 has a cache miss in its second execution cycle at B_1, and a cache miss latency of 3 cycles in the architecture. The sequential execution of the four threads in a machine with ideal memory would require 16 cycles, but it would spend 19 cycles in a real-memory machine due to the cache miss penalty. The execution time of the four threads by using interleaved multithreading can be reduced to 16 cycles, since

LC0	LC1	LC2	LC3
A_0	B_0	–	–
–	B_1	–	–
A_2	–	–	–
A_3	–	C_3	D_3

Thread 0

LC0	LC1	LC2	LC3
A_0	B_0	–	–
–	B_1	C_1	–
A_2	–	–	–
A_3	–	–	D_3

Thread 1

LC0	LC1	LC2	LC3
A_0	–	–	–
–	B_1	C_1	D_1
A_2	B_2	–	–
A_3	–	–	D_3

Thread 2

LC0	LC1	LC2	LC3
A_0	–	–	–
A_1	B_1	–	–
A_2	B_2	C_2	–
A_3	–	–	D_3

Thread 3

(a) **Logical Cluster Assignments done by Compiler**

PC0	PC1	PC2	PC3
A_0	B_0	–	–
–	B_1	–	–
A_2	–	–	–
A_3	–	C_3	D_3

Thread 0

PC0	PC1	PC2	PC3
–	A_0	B_0	–
–	–	B_1	C_1
–	A_2	–	–
D_3	A_3	–	–

Thread 1

PC0	PC1	PC2	PC3
–	–	A_0	–
C_1	D_1	–	B_1
–	–	A_2	B_2
–	D_3	A_3	–

Thread 2

PC0	PC1	PC2	PC3
–	–	–	A_0
B_1	–	–	A_1
B_2	C_2	–	A_2
–	–	D_3	A_3

Thread 3

(b) **Physical Cluster Assignments After Cluster Renaming**

	PC0	PC1	PC2	PC3
Cycle 0	A_0	B_0	A_0	A_0
Cycle 1	B_1	A_0	B_0	A_1
Cycle 2	C_1	D_1	–	B_1
Cycle 3	B_2	C_2	–	A_2
Cycle 4	–	B_1	B_1	C_1
Cycle 5	–	A_2	A_2	B_2
Cycle 6	A_2	D_3	A_3	–
Cycle 7	D_3	A_3	D_3	A_3
Cycle 8	A_3	–	C_3	D_3

(c) **Merged Instruction Stream with ideal memory**

	PC0	PC1	PC2	PC3
Cycle 0	A_0	B_0	A_0	A_0
Cycle 1	B_1	A_0	B_0	A_1
Cycle 2	C_1	D_1	–	B_1
Cycle 3	–	B_1	B_1	C_1
Cycle 4	A_2	A_2	A_2	B_2
Cycle 5	D_3	A_3	–	–
Cycle 6	–	D_3	A_3	–
Cycle 7	B_2	C_2	–	A_2
Cycle 8	A_3	–	C_3	D_3
Cycle 9	–	–	D_3	A_3

(d) **Merged Instruction Stream with a cache miss penalty**

Figure 5: **Instruction stream on a 4 clustered machine with 4 threads with round robin priority.**

the cache miss latency is hidden by the interleaving.

The effect of using CSMT in this case is shown in figures 5(c) and 5(d) for ideal and real memory respectively. For each cycle, the priority of thread assignment changes by following a round robin policy. Thus, cycle 0 starts by assigning bundles A_0 and B_0 from thread 0 to physical clusters 0 and 1. T1 cannot be scheduled, since operations in LC0 are assigned to PC1, and this cluster is already used by bundle B_0 from thread 0. So, bundles A0 of threads 2 and 3 are scheduled at PC2 and PC3 respectively, since no collision exists in the physical clusters assignment.

At cycle 1, the highest priority is assigned to operations belonging to thread 1. The mapping function assigns operations from LC0 to PC1 and operations from LC1 to PC2. Bundles from thread 2 cannot be assigned due to collision, and then operations from thread 3 are assigned to the free clusters. Operations from Thread 0 are neither scheduled since no cluster is available. The highest priority is assigned in next cycle to Thread 2, and so on.

Note that the assignment is at cluster level, so some FUs of each cluster can be idle despite the cluster being in use, because we consider a cluster is allocated when some operation is assigned to the respective bundle and the bundle may use only a few FUs if there is not enough ILP(The bundle may contain no-ops).

As figures 5(c) and 5(d) show, CSMT would require only 9 cycles in an ideal memory machine and 10 cycles when a cache miss penalty of 3 cycles is assumed, improving interleaving results by more than 40% in both cases. CSMT is explained in detail in section 4.

3 Base Architecture

The base architecture is based on the VEX clustered architecture which is modeled upon the HP/ST Lx [9] VLIW family. The VEX C compiler used in this study is a derivation of HP/ST ST220 C compiler, that uses *Trace Scheduling* [10] as global scheduling algorithm.

VEX is a 32 bit clustered Integer VLIW architecture and provides scalability of issue width and functionality. FUs within a cluster can access only local register files with the exception of Branch FU, which may read registers from other clusters. Clusters are architecturally visible and require explicit inter-cluster copy operations to move data across them. VEX is a *less-than-or-equal* (LEQ) machine, where operations latencies are exposed to the compiler and, if hardware can complete an operation in the same or fewer cycles, no deadlocks are required. However, for operations like memory accesses, which may take longer than the assumed latency, execution is stalled until the architectural assumptions hold true.

Each cluster has 2 multipliers and 1 load/store unit, and number of ALUs is same as issue width of the cluster. Memory and multiply have a latency of 2 cycles, and the rest have a single cycle latency.

There is no branch predictor and fall-through path is the predicted path. The incorrect instructions issued following a taken branch are squashed. Branches are two phased: the first part does the comparison and sets the branch registers ahead of the actual branch, and second is the actual control flow changing branch instruction. There is a delay of 2 cycles in the two phases, but this implementation allows to achieve a taken branch penalty of only 1 cycle if the 2-cycle delay can be filled with useful instructions.

A 64 KB 4-way set-associative, 64 byte line size cache structure is assumed for both ICache and DCache.

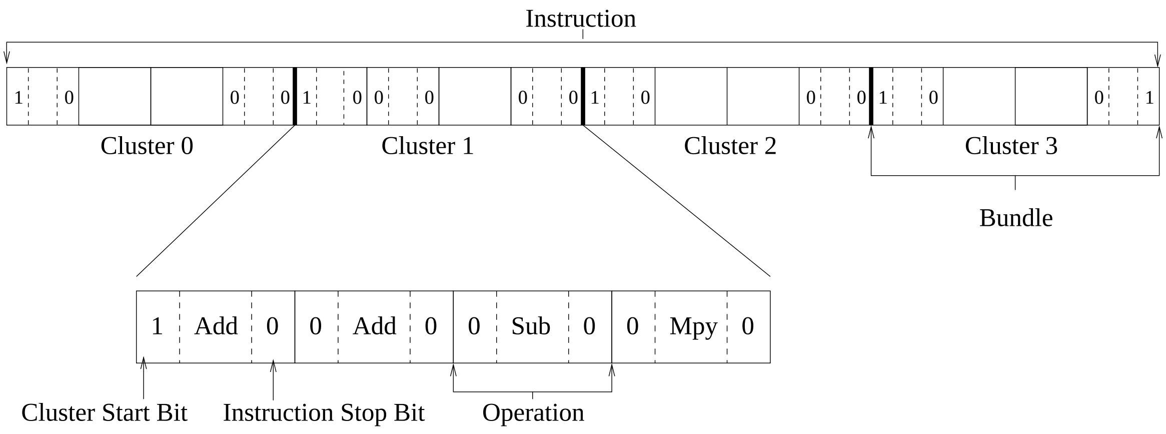


Figure 6: **A VEX Instruction**

To avoid storing nops in memory, VEX has two reserved bits for sequencing (instruction-stop bit which indicates end of instruction) and cluster encoding (cluster-start bit which indicates beginning of a new cluster) per operation. An operation is the basic execution unit, collection of operations scheduled to execute in the same cluster form a bundle and the collection of bundles scheduled to execute together is called an instruction. For a 4-issue cluster, a bundle contains between 1 and 4 operations. When the cluster-start bit is set to ’1’, the current operation and the following are assigned to a new cluster. When the instruction-sop bit is set to ’1’, the current instruction is finished and the following operations belong to the next instruction. Any instruction can have up to 4 cluster-start bit set to ’1’, but only the last operation of the instruction has the instruction-stop bit set to ’1’. The instruction breakup into bundles and operations is shown in Figure 6.

The VEX architecture is decoupled from the implementation of the inter-cluster communication networks and, for our evaluations, a point to point communication network has been used.

The architecture supports only integer applications and compiler supports floating point by emulation, which can cause a huge slowdown for floating point applications. Therefore, only integer applications have been considered for our evaluations.

4 CSMT Architecture

Our proposed CSMT architecture is built upon the base architecture, with some microarchitectural changes and extra hardware for multithreading. This section describes these changes which include our novel cluster renaming technique, changes in the pipeline and its effects, thread selection policy, support for precise exceptions and CSMT requirements.

4.1 Cluster Renaming

We have virtualized the cluster naming mechanism to achieve a dynamic renaming of clusters for the threads. The cluster renaming tries to maximize the distance between the logical *cluster 0* of different threads so that cluster conflicts between threads are minimized. The renaming function used is simply a cluster shift value for the threads, based on the number of clusters and the number of threads running at a particular moment so as to maximize the distance between the heavily used clusters, as shown in Equation (1). For instance, for 2 threads on a 2 cluster machine, 1 can be used as a shift value between threads. On a 4 cluster machine, the shift value can be kept to 2 for 2 threads, so while logical cluster 0 on thread 0 will still map to physical cluster 0, logical cluster 0 on thread 1 will be renamed as physical cluster 2.

$$Shift = Thread\_Id \times \begin{cases} \left\lfloor \dfrac{Number\ of\ Clusters}{Number\ of\ Threads} \right\rfloor, & (Number\ of\ Clusters \geq Number\ of\ Threads) \\ 1, & (Number\ of\ Clusters < Number\ of\ Threads) \end{cases} \tag{1}$$

Figure 7 shows the effect of cluster renaming for 2 threads on a 4 clustered architecture. Label c{x} means the instruction is scheduled in logical cluster x, \$r{y}.{z} means[2] register z in cluster y and ’; ;’ indicates end of the VLIW instruction. Before renaming, both of the threads use cluster 0 and 1, since not enough parallelism

[2]Though we show the cluster number in the register name for more clarity, it is not used in the actual instruction encoding, as that information is already available by the cluster name

was available to use clusters 2 and 3. After cluster renaming, thread 1 uses physical clusters 2 and 3 instead of the originally scheduled clusters and now, the instructions from both threads can be executed simultaneously. Note that, besides renaming the execution cluster, it is also required to rename the operands for intercluster communication.

```
c0 add $r0.2 = $r0.3, $r0.0
c0 add $r0.5 = $r0.3, $r0.1 ;;
c0 add $r0.4 = $r0.2, $r0.3
c1 sub $r1.4 = $r1.3, $r1.5 ;;
```

(a) **Thread 0**

```
c0 mul $r0.1 = $r0.7, $r0.13
c0 shl $r0.6 = $r0.3, 4
c1 add $r1.2 = $r1.7, $r1.3 ;;
c0 add $r0.4 = $r0.6, $r0.8 ;;
```

(b) **Thread 1**

```
c2 mul $r2.1 = $r2.7, $r2.13
c2 shl $r2.6 = $r2.3, 4
c3 add $r3.2 = $r3.7, $r3.3;;
c2 add $r2.4 = $r2.6, $r2.8;;
```

(c) **Modified code sequence for thread 1**

Figure 7: **Code sequence for two threads on a 4 cluster machine.**

The same effect could have been produced by the compiler without any need for doing it in hardware. However, that would require the compiler to know all the applications that will run simultaneously in a multithreaded environment.

Figure 8 shows the shift and mapping tables for 4 threads on a 4 clustered architecture.

Thread Id	*Shift*
0	0
1	1
2	2
3	3

(a) **Shift Table**

	Logical Clusters			
	0	**1**	**2**	**3**
	Physical Mapping			
Thread 0	0	1	2	3
Thread 1	1	2	3	0
Thread 2	2	3	0	1
Thread 3	3	0	1	2

(b) **Virtual Cluster Table**

Figure 8: **Shift and Mapping Tables for 4 threads on a 4 cluster machine**

This mapping is only visible to the processor and transparent to the compiler. This avoids any special compilation to achieve extra performance on a multithreaded platform. Also, the shift is dynamic but fixed for a thread once it starts executing until it finishes or is switched out of context. This allows a low complexity hardware implementation.

4.2 Processor Configuration

The special bits in instruction encoding to indicate start of new cluster and end of instruction make it straightforward to detect which clusters are in use by having a partial decoding. A combinational logic implementation for collision detection and instruction merging, coupled with a set of multiplexers, allows a fast and cheap

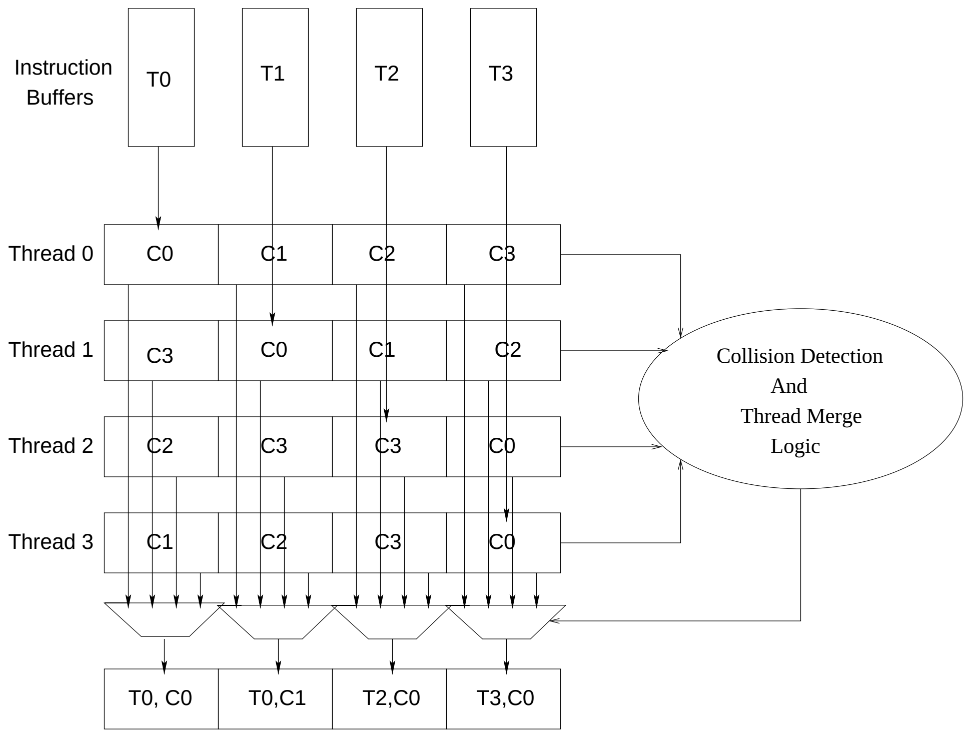


Figure 9: **Thread Merging Hardware**

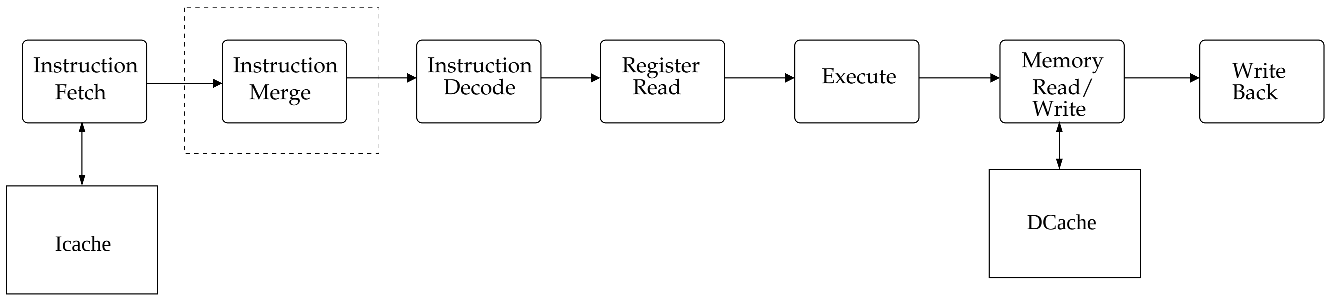


Figure 10: **Processor Pipeline**

hardware implementation. Figure 9 shows the thread merging hardware. Assuming the same scenario of cycle 0 in Figure 5, the collision detection and merge logic detects that thread 1 cannot be scheduled as there is a collision between it and the higher priority thread 0, but instructions from thread 2 and 3 do not collide. It generates appropriate select logic for the multiplexers, so that the instructions from the non-colliding threads can be issued.

To prevent any negative effect on the cycle time, this hardware is moved into a separate pipeline stage. Hence, the pipeline we use is similar to Lx pipeline except for an extra pipeline stage for instruction merging, as shown in Figure 10. There is a 2-cycle taken branch penalty, which is 1 more than Lx, to accommodate the extra pipeline stage. The incorrect instructions issued following a taken branch are squashed automatically by the processor.

Each pipeline stage is tagged with individual thread identifiers. This tagging is used to selectively flush instructions from a particular thread at a branch misprediction or a cache miss.

Each thread has its own register file per cluster, as shown in Figure 11, which can be an important design

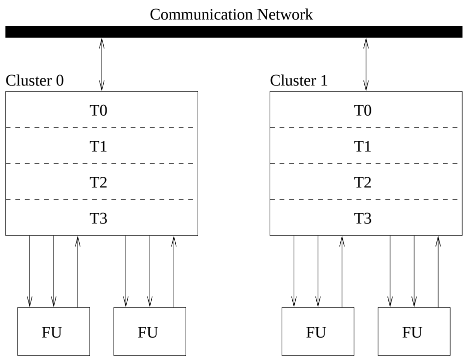


Figure 11: **CSMT Register File for 2 Threads**

consideration. Register file access time grows linearly with the number of ports but logarithmically with number of registers, and area grows quadratically with the number of ports. Then, separate register file for clusters are significantly cheaper and faster than having a big monolithic register file with the same number of ports and registers. This cost is present in all multithreading designs we are aware of.

4.3 Thread Selection Policy

The execution packet is formed by merging instructions from as many threads as possible. Conflicts are resolved on the basis of their individual priority. Each cycle, a different priority is assigned to each thread in a round robin fashion. The instructions from the highest priority thread are selected; then the next priority thread is selected if it does not collide with the already formed packet and so on.

Round robin priority policy is in general the best policy , since it guarantees fairness and avoids starvation. However, for meeting the demands of applications which have real time or Quality of Service(QoS) requirements, it is possible to have fixed priority levels for threads which could be exposed and controlled by the Operating System. Since we assume a more general usage, we have evaluated only the round robin policy.

4.4 Exception Detection and Recovery

No special extra hardware is required for exception detection because of the in-order pipeline and thread tagging done at each pipeline stage. So, if an exception is detected at any time, it is straightforward to know which thread and what instruction caused the exception. On an exception, the pipeline of the faulting thread is flushed and exception handler is invoked.

4.5 CSMT Requirements

For our CSMT implementation, we require all clusters to be homogeneous. This is usually true for most of the existing clustered VLIW processors and, in particular, for Lx. The only exception in Lx is branch unit, that has to be replicated in all the clusters. All the FUs have to be fully pipelined as well so that the instructions from other threads can use that FU next cycle, since we do not track resource unavailability because of non-pipelined FUs. A LEQ (*less-than-equal*) model of execution, as previously explained in Section 3, is also necessary so that delay in issuing the next instruction can be tolerated and exceptions can be dealt with. Also, since we have assumed a point to point communication, communication conflicts do not arise. Evaluation and implementation over other communication networks are part of future research.

5 Performance Evaluation

Benchmarks	*ILP Degree*	*Description*	ILP_r	ILP_p
181.mcf	L	Minimum Cost Flow	0.96	1.34
256.bzip2	L	Bzip2 Compression	0.81	0.83
blowfish	L	Encryption	1.11	1.47
gsmencode	L	GSM Encoder	1.07	1.07
g721encode	M	G721 Encoder	1.75	1.76
g721decode	M	G721 Decoder	1.75	1.76
cjpeg	M	Jpeg Encoder	1.12	1.66
djpeg	M	Jpeg Decoder	1.76	1.77
imgpipe	H	Imaging pipeline	3.81	4.05
colorspace	H	Colorspace Conversion	5.47	8.88

Table 1: **Benchmarks**

Config	ILP Comb	Bench 1	Bench 2
21	L M	mcf	g721decode
22	L L	bzip2	blowfish
23	L H	bzip2	colorspace
24	L M	bzip2	djpeg
25	L H	bzip2	imgpipe
26	M M	cjpeg	g721decode
27	M M	djpeg	g721encode
28	M H	djpeg	imgpipe
29	M H	g721decode	colorspace
210	H H	imgpipe	colorspace

Table 2: **2-Thread Configurations**

Config	ILP Comb	Bench1	Bench2	Bench3	Bench4
41	L L H H	mcf	bzip2	colorspace	imgpipe
42	L L M H	mcf	bzip2	g721decode	imgpipe
43	L H M H	mcf	colorspace	djpeg	imgpipe
44	L H M H	mcf	colorspace	g721decode	imgpipe
45	L M M H	mcf	djpeg	g721decode	colorspace
46	L M M H	mcf	djpeg	g721decode	imgpipe
47	L L M M	bzip2	mcf	djpeg	g721decode
48	L L M H	bzip2	mcf	djpeg	imgpipe
49	L L M H	bzip2	mcf	g721decode	colorspace
410	L M M H	bzip2	djpeg	g721decode	colorspace
411	L H M H	bzip2	imgpipe	g721decode	colorspace

Table 3: **4-Thread Configurations**

In order to test the efficacy of CSMT, we have used a set of MediaBench [14] and relevant SpecInt 2000 [11] applications, as tabulated in Table 1. We have also included a production color space conversion benchmark

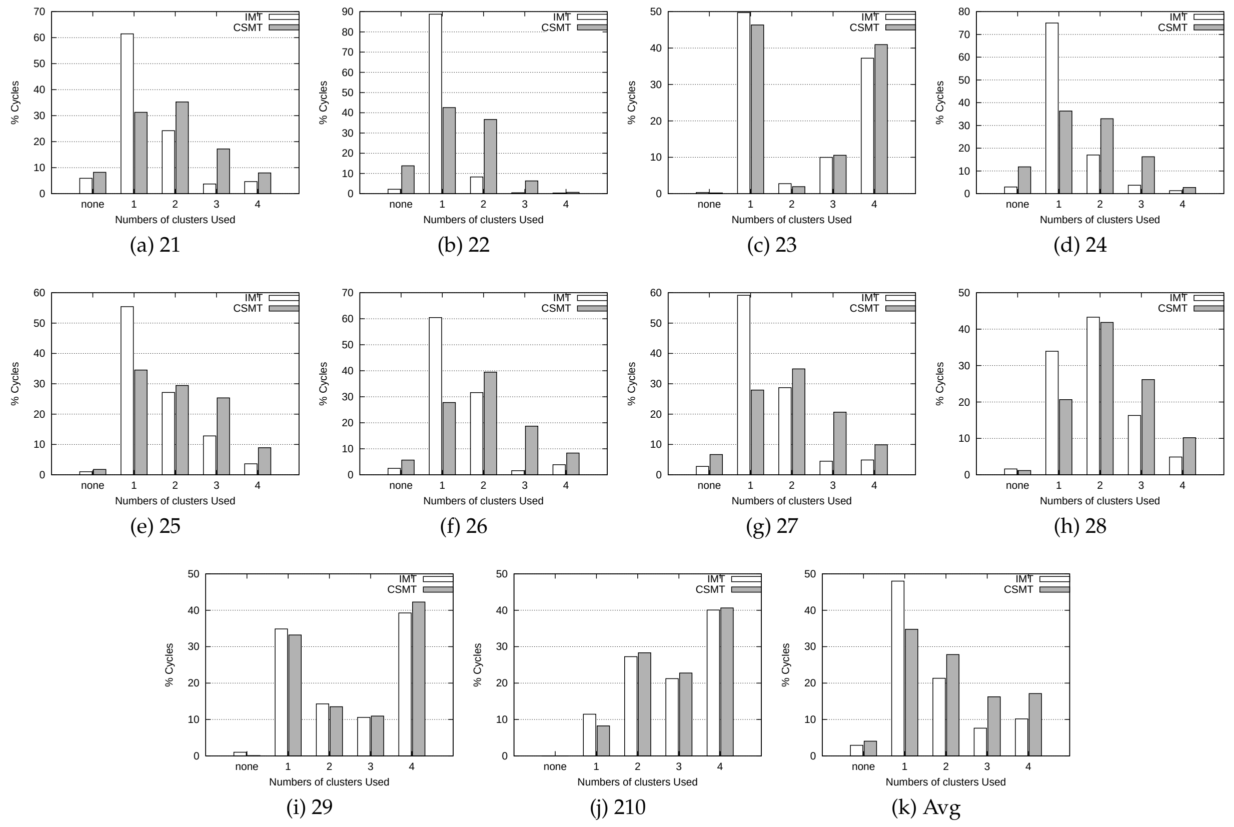


Figure 12: **Cluster Usage with 2 Threads, Perfect Memory**

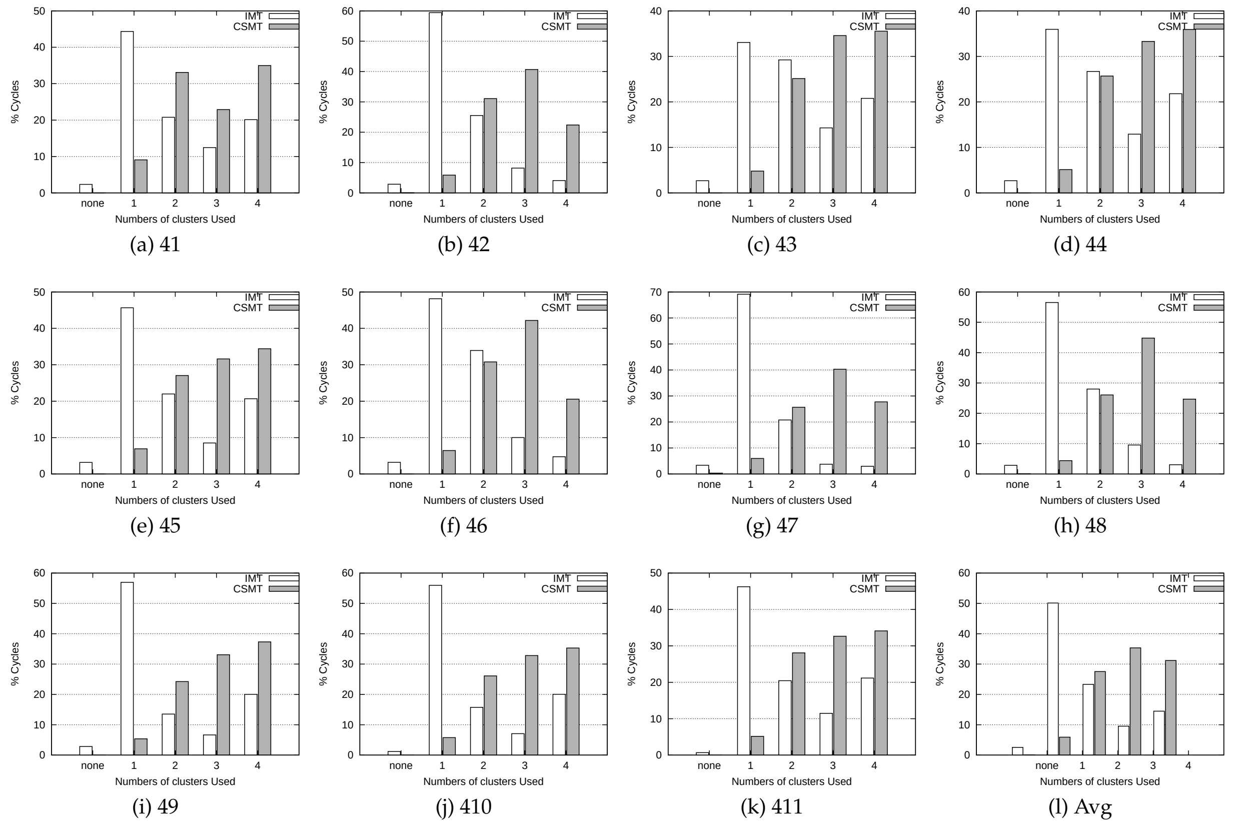
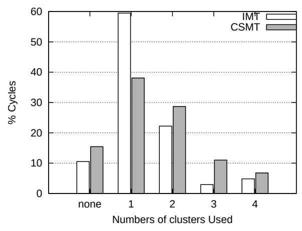
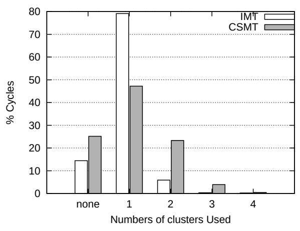


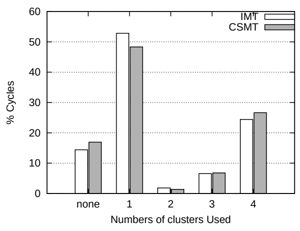
Figure 13: **Cluster Usage with 4 Threads, Perfect Memory**



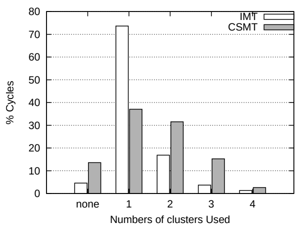
(a) 21



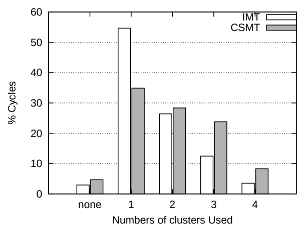
(b) 22



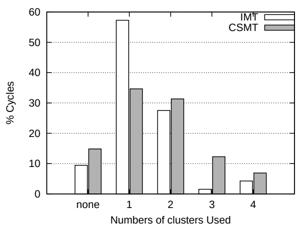
(c) 23



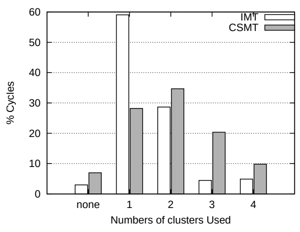
(d) 24



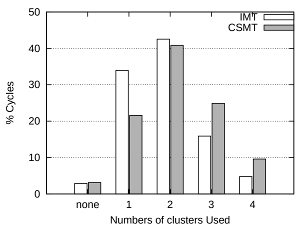
(e) 25



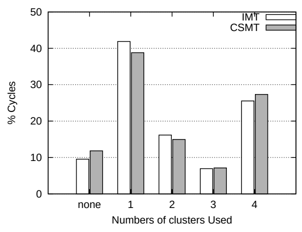
(f) 26



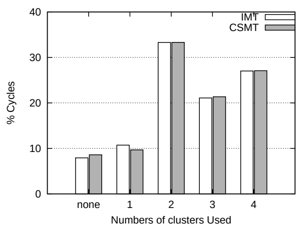
(g) 27



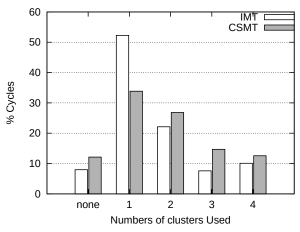
(h) 28



(i) 29



(j) 210



(k) Avg

Figure 14: **Cluster Usage with 2 Threads, Real Memory**

used in high performance printers [1] in our experiments.

Experiments have been done in a 4 issue, 4-cluster architecture configuration by using two and four threads respectively. All the experiments have also been done both by assuming an ideal memory model with no cache misses and the real memory model discussed previously in Section 4. Columns ILP_r and ILP_p in Table 1 show, for each benchmark, the ILP calculated for real and perfect memory models respectively. Benchmarks can be classified by using their ILP into three categories: high ILP (as colorspace and imgpipe), medium ILP (as g721encode, g721decode, cpeg and dpeg) and low ILP (181.mcf, 256.bzip2, blowfish and gsmencode). This classification is shown in column $ILP\ Degree$ as L (low ILP), M (Medium ILP) and H (High ILP).

The set of configurations used for multithreading is listed in tables 2 (for 2 threads) and 3 (for 4 threads). In order to select every thread configuration, we have combined benchmarks with different ILP degrees, attempting to cover all possible combinations. Column labeled as $ILP\ Comb$ indicates these ILP combinations. For example, configuration 41 in Table 3 has two benchmarks with low ILP and two benchmarks with high ILP (LLHH), configuration 47 has two benchmarks with low ILP and two benchmarks with medium ILP (LLMM) and configuration 411 has one benchmark with low ILP, one benchmark with medium ILP and two benchmarks with high ILP (LHMH).

In order to compare CSMT to other techniques previously proposed in the literature, we have evaluated the performance obtained by using a fine grained interleaved multithreading model (IMT) [20, 3]. The architectural parameters are the same for interleaved multithreading as for CSMT, except that the extra pipeline stage is not

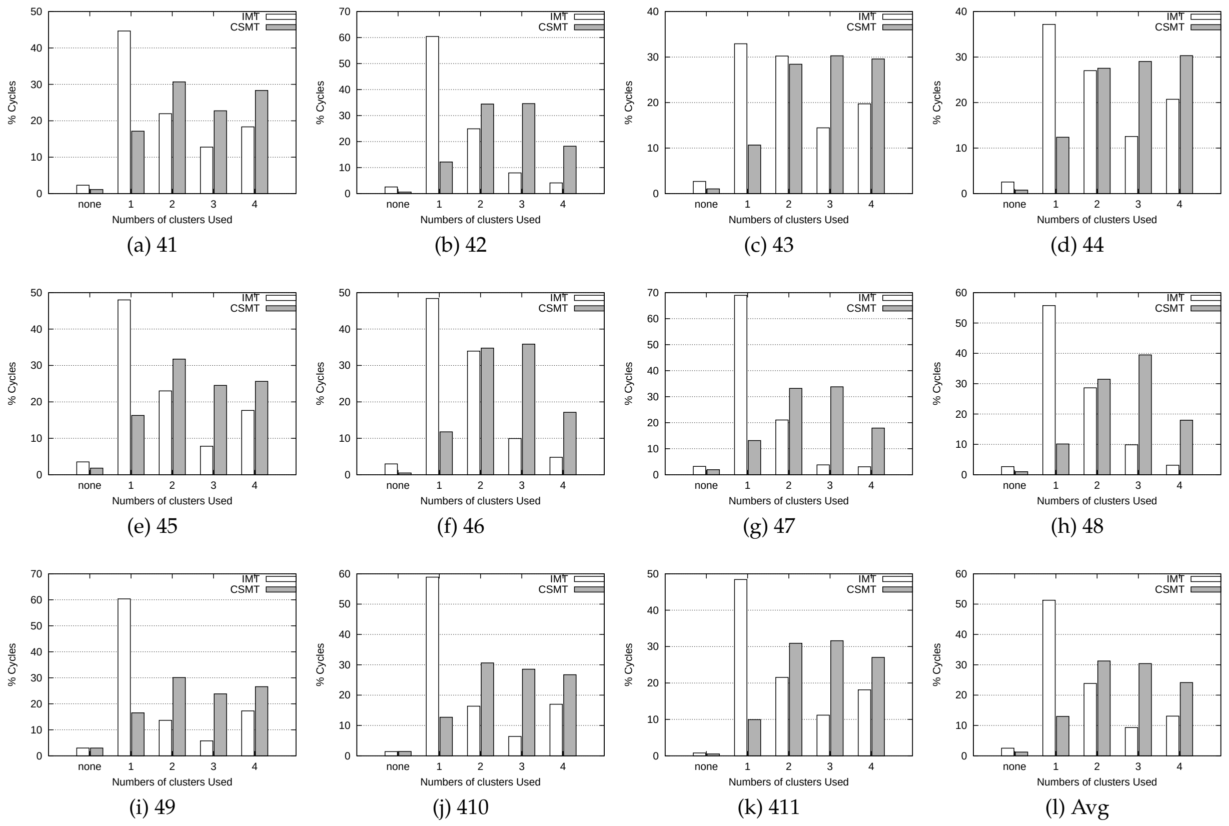


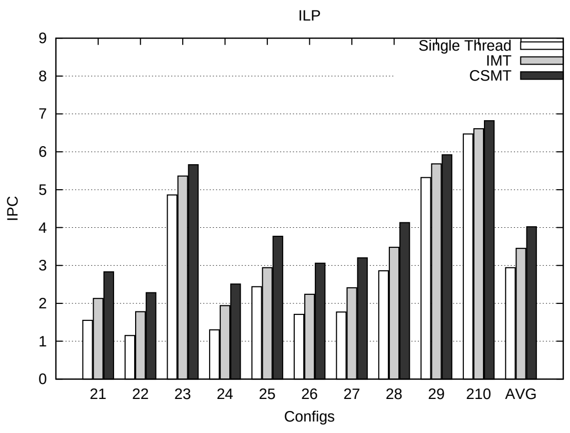
Figure 15: **Cluster Usage with 4 Threads, Real Memory**

there and the taken branch penalty is 1 cycle instead of 2, as assumed for CSMT.

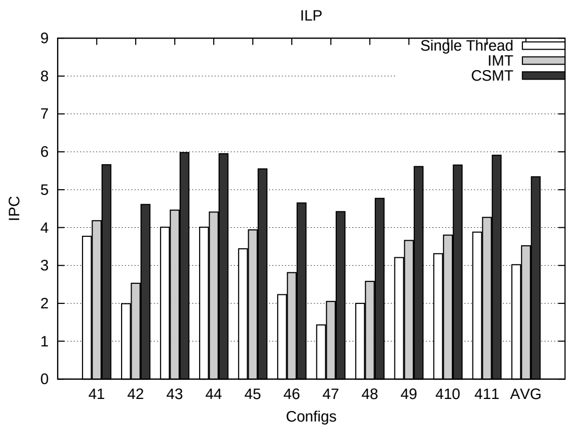
Figures 12 and 13 show the cluster usage statistics by assuming a perfect memory model with no cache misses and by using 2 and 4 threads respectively. As can be seen, the cluster usage improves considerably with CSMT, specially when increasing the number of threads, since many clusters are used most of the time. On average, the improvement with 2 threads is moderate because of the limited ILP available. However, when 4 threads are used the average improvement in cluster usage is spectacular with a extremely low percentage of cycles where 1 or none clusters are used, using 3 or 4 clusters more than 65% of the time. It is interesting to note that, with a low number of high ILP threads like in mix 210, there is practically no difference in cluster usage between IMT and CSMT. This is because there are very few opportunities to combine instructions from two threads if both already use a high number of clusters.

Figures 14 and 15 show the cluster usage statistics when a real memory model is used (see Section 4), with 2 and 4 threads respectively. Notice that with real memory the improvement is not so significant for 2 threads, though CSMT still does better. This is because now most of the time the inter-thread parallelism is used to hide cache miss stall cycles. On the other hand, with 4 threads CSMT still does significantly better than IMT.

Finally, we have computed the ILP achieved for each configuration by CSMT and IMT. We have also calculated the ILP achieved in a single threaded machine in order to compare both approaches to it. Figures 16(a) and (b) show the results obtained when running simultaneously 2 and 4 threads respectively by assuming perfect memory, and figures 17(a) and (b) show the same results when assuming the real memory model detailed

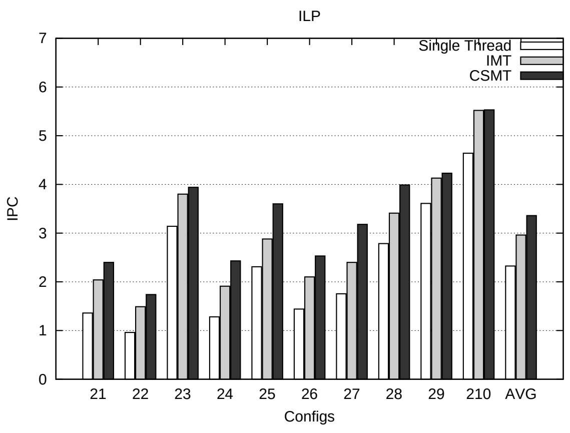


(a) **2 Threads**

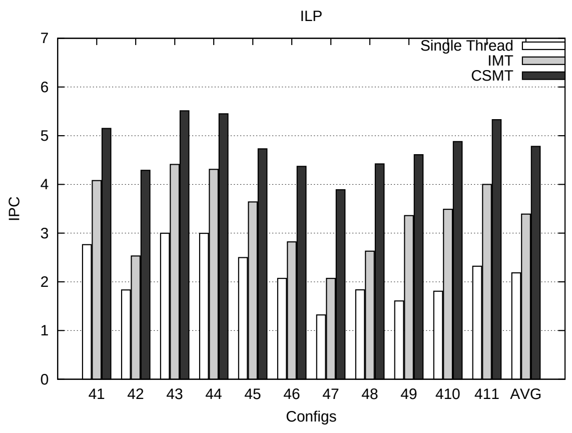


(b) **4 Threads**

Figure 16: **IPC with 4 clusters, Perfect Memory**



(a) **2 Threads**



(b) **4 Threads**

Figure 17: **IPC with 4 clusters, Real Memory**

in Section 4.

First thing to notice is that, with an ideal memory, IMT does slightly better than the single threaded machine. This is because, despite there are no vertical no-ops due to memory stalls, a few issue cycles are lost due to taken branches and def to use latency of operations like loads, multiplies and compares. IMT can easily hide those vertical no-ops by issuing instructions from an alternate thread. Since SMT can also hide horizontal no-ops, it clearly outperforms IMT, specially with 4 threads. In this case, CSMT has an average speedup of 76% over a single threaded VLIW and 51% over IMT. Notice that, for low ILP benchmark mixes like 47, the speedups are as high as 209% over single thread and 115% over IMT.

Finally, when real memory is considered, the performance of a single thread VLIW is significantly degraded, while IMT and CSMT only suffer a minor impact, which shows the ability of both approaches to hide vertical no-ops. CSMT, however, still outperforms IMT by a significant margin. For instance, with 4 threads CSMT shows an average speedup of 118% over single thread and 41% over IMT. Notice that, in particular cases like mix 47, speedup can be as high as 88% over IMT and 195% over single thread.

6 Conclusions

In this paper we have presented CSMT, a new Cluster-level Simultaneous MultiThreading approach for VLIW clustered processors. The analysis performed on a set of benchmarks shows that, in general, no cluster is used during a significant amount of time due (mostly) to cache misses. Moreover, most applications have low ILP and use only a few clusters most of the time.

The compiler assigns always first clusters to all threads and tries to reduce the number of clusters used by each thread in order to also reduce communication overhead among different clusters. As a consequence, the assigned clusters collide when several threads are executed simultaneously. CSMT avoids this problem and allows a more parallel execution of the threads by renaming the clusters previously assigned by the compiler. The renaming mechanism is fast and has a very low hardware complexity.

Our results show that CSMT makes a better use of clusters than interleaved multithreading (IMT). In terms of performance, CSMT clearly outperforms IMT. For instance, for a 4-cluster machine with 4 threads, CSMT shows an average speedup of 75% over a single threaded machine and of 53% over IMT assuming no cache misses, which shows the ability of CSMT to hide horizontal no-ops. When a realistic memory system is assumed, the speedup over single thread increases to 118% while speedup over IMT is reduced to 41%. This is because now most of the resources wasted are due to stalls caused by cache misses, and IMT already does a very good job hiding memory latency. However, CSMT still has a very significant advantage over IMT due to its ability to hide both vertical and horizontal no-ops.

References

[1] Colorspace conversion program used in high performance printers, personal communication.

[2] Anant Agarwal, Jonathan Babb, David Chaiken, Godfrey D'Souza, Kirk L. Johnson, David A. Kranz, John Kubiatowicz, Beng-Hong Lim, Gino Maa, Ken Mackenzie, Daniel Nussbaum, Mike Parkin, and Donald Yeung. Sparcle: A multithreaded vlsi processor for parallel computing. In *Parallel Symbolic Computing*, pages 359–361, 1992.

[3] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton J. Smith. The tera computer system. In *ICS*, pages 1–6, 1990.

[4] Domenico Barretta, William Fornaciari, Mariagiovanna Sami, and Daniele Bagni. Multithreaded extension to multicluster vliw processors for embedded applications. In *DATE*, pages 748–749, 2005.

[5] Ramon Canal, Joan-Manuel Parcerisa, and Antonio González. Dynamic cluster assignment mechanisms. In *HPCA*, 2000.

[6] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth, and Paul K. Rodman. A vliw architecture for a trace scheduling compiler. In *ASPLOS-II*, pages 180–192, 1987.

[7] StarCore DSP. http://www.starcore-dsp.com.

[8] TI TMS320C64xx DSPs. www.ti.com.

[9] Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, Giuseppe Desoli, and Fred Homewood. Lx: A technology platform for customizable vliw embedded processing. In *ISCA*, pages 203–213, 2000.

[10] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Computers*, 30(7):478–490, 1981.

[11] John L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *IEEE Computer*, 33(7):28–35, 2000.

[12] Intel. Intel itanium architecture software developer's manual, 2002.

[13] Bharath Iyer, Sadagopan Srinivasan, and Bruce L. Jacob. Extended split-issue: Enabling flexibility in the hardware implementation of nual vliw dsps. In *ISCA*, pages 364–375, 2004.

[14] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *MICRO*, pages 330–335, 1997.

[15] SUN MAJC. www.sun.com/microelectronics/majc.

[16] Alfred Mikschl and Werner Damm. Msparc: A multithreaded sparc. In *Euro-Par, Vol. II*, pages 461–469, 1996.

[17] Philips. http://www.semiconductors.philips.com/trimedia/.

[18] B. Ramakrishna Rau. Dynamically scheduled vliw processors. In *MICRO*, pages 80–92, 1993.

[19] B. Ramakrishna Rau, David W. L. Yen, Wei C. Yen, and Ross A. Towle. The cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs. *IEEE Computer*, 22(1):12–35, 1989.

[20] B. J. Smith. Architecture and applications of the hep multiprocessor computer system. In *SPIE*, pages 241–248, 1981.

[21] Marc Tremblay, Jeffrey Chan, Shailender Chaudhry, Andrew W. Conigliaro, and Shing Sheung Tse. The majc architecture: A synthesis of parallelism and scalability. *IEEE Micro*, 20(6):12–25, 2000.

[22] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA*, pages 392–403, 1995.