Cluster-Level Simultaneous Multithreading for VLIW Processors

Manoj Gupta* **Fermín Sánchez** **Josep Llosa**
Department of Computer Architecture
Universitat Politécnica de Catalunya
{mgupta,fermin,josepll}@ac.upc.edu

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 wide 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. CSMT considers the set of operations that execute simultaneously in a given cluster (named bundle) as the assignment unit. All bundles belonging to a VLIW instruction from a given thread are issued 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 113% over a single-thread VLIW architecture and 36% over Interleaved MultiThreading (IMT). In some cases, speedup can be as high as 228% over single thread architecture and 97% over IMT.

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 superscalar processors, no special hardware like register renaming, instruction queues, reorder buffers, etc. is required. VLIWs have been used in general purpose computing [3, 16, 10]. However, due to the hardware simplicity, low cost and low power consumption, the VLIW paradigm has found its niche in embedded computing [5, 18].

Many embedded applications exhibit significant amounts of ILP, or at least regions with high ILP interleaved with low ILP regions. 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 [17]. The 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. Higher issue width allows to achieve higher performance levels without scaling up the frequency achieving a better power budget as well. Many VLIWs have been designed using the clustered approach [9, 18].

Many applications scale well with issue width, for instance, colorspace conversion used in high performance printers has an IPC of 3.9, 6.0 and 8.9 for an issue width of 4, 8 (2 clusters of 4) and 16 (4 clusters of 4) respectively, which makes a very high issue width processor desirable. However, the ILP exposed in many applications, or in some code regions, is limited and the processor is heavily underutilized. Also, in a production environment, high ILP applications like image processing coexist with low ILP applications like control code or the OS itself. Simultaneous MultiThreading (SMT) [20] is a well known technique to improve the resource utilization by exploiting thread level parallelism (TLP). Pure SMT at the operation level has been proposed for VLIWs [11, 14]. [14] is aimed at increasing single program performance using compiler-generated

*This work is supported by Spanish Ministry of Science and Technology under contract TIN2004-07739-C02-01, SARC (Scalable computer ARChitecture) Project and HiPEAC European Network of Excellence

1-4244-1258-7/07/$25.00 ©2007 IEEE

(speculative) threads in a multithreaded VLIW architecture. It involves extensive changes to the ISA, compiler support for generation of multiple threads, and additional hardware resources such as: buffers for speculative load and store instructions, a thread synchronization hardware and a complex operation welder. The welder is implemented as a crossbar and, because of the complex hardware, scalability is limited beyond two threads for cost/performance reasons. [11] requires some sort of out-of-order execution [15], 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 in the literature.

Block multithreading [22] executes instructions from a single thread until it is blocked by an event (a cache miss, for instance). 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[1] wasted due to context switch time. Moreover, no horizontal empty slot[2] inside a VLIW can be used for other threads.

Interleaved multithreading [19] does a zero cycle context switch every cycle, so that 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. Moreover, it still does nothing to remove horizontal waste.

In [2], 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. 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 from different threads when no conflict exists in the clusters they require to execute. In order to reduce cluster conflicts when several threads require the same cluster, a very simple hardware-based 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 is removed. In addition,

[1]Cycles where no operations are issued

[2]Operation slot inside a VLIW with a nop

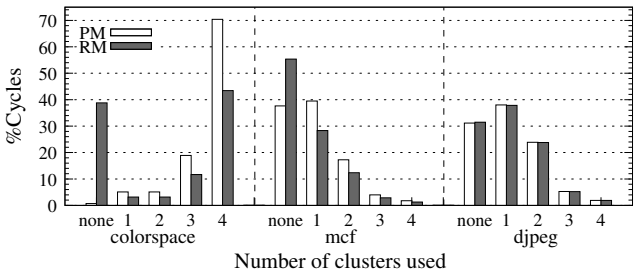

Figure 1: Cluster usage

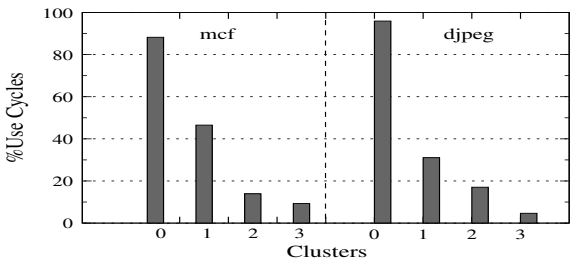

Figure 2: Individual cluster use in mcf and djpeg

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.

The rest of the paper is organized as follows. Section 2 presents some statistics that motivate CSMT, as well as a simple example. The details of the CSMT architecture are discussed in Section 3. An evaluation and comparison with other approaches is performed in Section 4. Finally, Section 5 concludes the paper.

2. Motivation

Figure 1 shows the percentage of time a given number of clusters are used in a 4-cluster architecture with 4-issue width per cluster respectively for the benchmarks colorspace [1], mcf [8] and djpeg [12]. We assume a cluster is used when any of its FUs is used. The figure presents data for a perfect memory model (PM) with no cache misses and a real memory model (RM). We have considered a 64-KB 4-way set-associative ICache and DCache and a cache miss latency of 20 cycles in this simulation for the real memory model. More details about the architecture are available in Section 3.

Colorspace[3] benchmark has a high ILP degree, close to 6 with a 8-issue width and close to 9 with a 16-issue width. Due to this high ILP, with a perfect memory model all the clusters are simultaneously in use most of the time. However, when a real memory model is considered, a significant amount of time is wasted in handling the cache misses

[3]Used in high performance printers

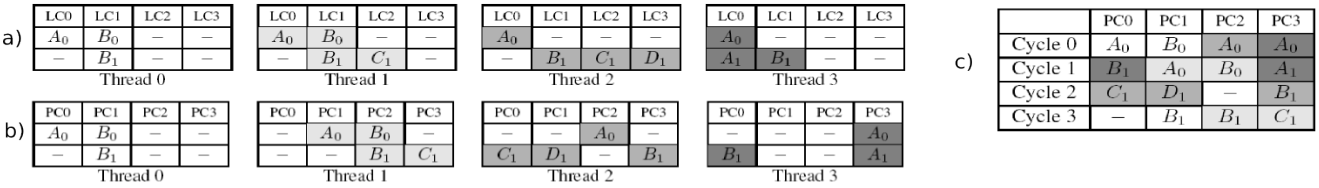

Figure 3: Instruction stream on a 4-thread 4-cluster architecture (a) Logical cluster assignments (b) Physical cluster assignments after cluster renaming (c) Merged instruction stream

(close to 40% of the execution). In order to reduce the time the processor is idle, a simple multithreading technique like interleaved multithreading [19] can be used to tolerate the cache misses by scheduling other threads during the miss intervals, increasing in that way the processor throughput.

Other applications exhibit much lower ILP. For instance, the average IPC in SPEC [8] or Mediabench [12] benchmarks range, in general, between 1 and 2. Thus, when repeating the previous experiment using SPEC and Mediabench benchmarks, we obtain significantly different results in cluster usage, as shown in Figure 1. A considerable amount of time is spent in cycles where no cluster is used. This time can be easily reduced by using interleaved multithreading (IMT). However, when any cluster is used, the cluster usage is very unbalanced and most of the time only a single cluster is used. This is reasonable, since the compiler tries to schedule as many operations as possible in a single cluster to avoid communication overhead. Only a small number of clusters thus are used most of the time, since there is not always enough ILP available during the program's execution.

Figure 2 represents the percentage of cycles in which each cluster is in use (whenever any cluster is used) for mcf and djpeg benchmarks. 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 both programs.

The use of SMT may improve the cluster utilization but, if we implement SMT in a naive way, most of the threads will compete for resources on a few clusters most of the time, rather than using the 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 easily derived from Figure 2.

CSMT, the SMT approach proposed in this paper, is based on renaming at execution time the clusters initially assigned by the compiler. Conflicts are resolved at cluster level instead of FU level, which is significantly easier and cheaper than a fully blown SMT. For example, if two threads (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. Cluster renaming is explained later in Section 3.1.

Figure 3 shows a sample multithreaded execution for a 4-thread 4-cluster architecture using CSMT. Figure 3(a) the cluster assignment (logical clusters) done by the compiler for the 4 threads. Letters A to D represent a bundle[4] scheduled in logical clusters 0 to 3 (LC0-LC3) and the subscript indicates the execution cycle in a single-thread environment. So, A_0 means the group of operations belonging to bundle A that are assigned to cluster 0 by the compiler and executed at cycle 0, B_0 the operations assigned to cluster 1 and executed at cycle 0, and so on. Figure 3(b) shows the physical mapping of clusters (PC0-PC3) done by CSMT after cluster renaming for each thread. Notice that this cluster renaming consists simply in rotating the original clusters by a fixed value for each thread.

The effect of using CSMT is shown in Figure 3(c). Thread priority follows a round robin policy. Initially, Thread 0 has the highest priority. Thus, cycle 0 starts by assigning bundles A_0 and B_0 from Thread 0 to physical clusters 0 and 1 respectively. Thread 1 cannot be scheduled, since cluster PC1 is already used by bundle B_0 from Thread 0. Bundles A_0 of threads 2 and 3 are scheduled at clusters 2 and 3 respectively, since no collision exists in the physical clusters assignment.

At cycle 1, the highest priority is assigned to operations belonging to Thread 1 following the round robin policy. Bundles A_0, B_0 from Thread 1 are assigned to clusters 1 and 2. Bundles from Thread 2 cannot be scheduled due to collision, and then bundles from Thread 3 are assigned to the free clusters. Operations from Thread 0 are not scheduled since no cluster is available. The highest priority is assigned in next cycle to Thread 2, and so on. The sequential execution of the four threads in a machine with perfect memory would require 8 cycles. CSMT, however, would require only 4 cycles, as shown in Figure 3(c).

3. CSMT Architecture

The CSMT architecture evaluated in this paper is based on the VEX clustered architecture [21] modeled upon the commercial HP/ST Lx [5] VLIW family. The VEX C compiler [21] used in this study is a derivation of HP/ST ST200 C compiler, which itself is a derivative of Multiflow compiler [13] that uses *Trace Scheduling* [6] as global schedul-

[4] 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 a VLIW instruction

ing algorithm and *Bottom Up Greedy* [4] as cluster assignment algorithm.

VEX is a 32-bit clustered integer VLIW architecture that 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) architecture, where 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 the number of ALUs is the same as the issue width of the cluster. Memory and multiply operations have latency of 2 cycles, and the rest have 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 operation does the comparison and sets the branch registers ahead of the actual branch, and the second is the actual control flow changing branch operation. There is a 2-cycle delay from compare to branch, and the taken branch penalty is 1 cycle.

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

The architecture supports only integer applications since it is based on the commercial ST200 [9] which has been used for many multimedia applications coded in fixed point arithmetic. Compiler supports floating point by emulation, which can cause a huge slowdown for floating point applications. For this reason, only integer applications have been considered in our experiments.

The proposed CSMT architecture is built upon the base VEX architecture with addition of extra hardware for multithreading and some CSMT specific microarchitectural changes.

Each thread has its own register file per cluster, which can be an important design consideration. 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. For 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 the branch unit, which need to be present in all clusters to have homogeneous clusters. The cost of the extra branch units however, is insignificant due to the simplicity of the unit. All the FUs have to be fully pipelined so that the instructions from other threads can use that FU next

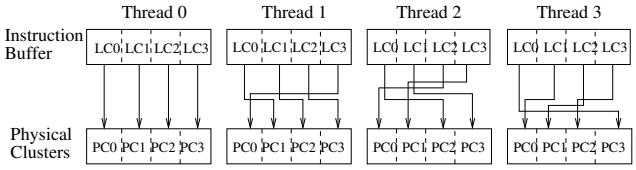

Figure 4: Cluster renaming logic for a 4-thread 4-cluster architecture

cycle, since we do not track resource unavailability because of non-pipelined FUs. A LEQ (*less-than-equal*) model of execution, is also necessary so that any delay in issuing the next instruction can be tolerated and exceptions can be dealt with. Also, since we have assumed a fully connected point to point communication, communication conflicts do not arise. No special or 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. When an exception occurs, the pipeline of the excepting thread is flushed and exception handler is invoked.

Next sections describe the microarchitectural changes, which include CSMT cluster renaming technique, thread merge hardware, changes in the pipeline and its effects.

3.1. Cluster Renaming

CSMT virtualizes the cluster naming mechanism to achieve a dynamic renaming of clusters for the threads. The cluster renaming distributes the same *Logical Cluster* of different threads to different *Physical Clusters* so that cluster conflicts between threads are reduced. The mapping is fixed for each thread once it starts executing until it finishes or is switched out of context. The renaming function used is simply a cluster shift value for each thread, based on the number of clusters and the number of simultaneous threads supported by the architecture. For instance, in a 4-thread 4-cluster architecture, Thread 0 is shifted by 0, Thread 1 is shifted by 1, Thread 2 is shifted by 2 and Thread 3 is shifted by 3. So, while logical cluster 0 on Thread 0 still maps to physical cluster 0, logical cluster 0 on Thread 2 will map to physical cluster 2. Similarly, for a 2-thread 4-cluster machine, a cluster shift value of 2 can be used.

The 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.

As the shift is fixed for each thread, it can be easily hardcoded in the hardware for a given number of threads and clusters. A very cheap cluster rename logic is possible by rerouting the wires from the instruction buffer of the threads to a hardcoded cluster. Figure 4 shows the wiring for a 4-thread 4-cluster processor. LCi means logical cluster i and PCi means physical cluster i.

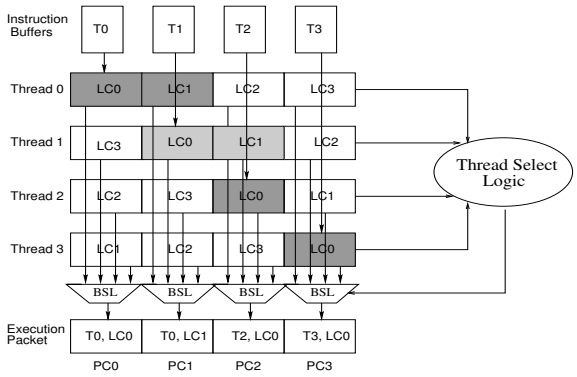

Figure 5: Thread merge hardware

The same effect could have been produced by the compiler. However, that would require the compiler to know all the applications that will run simultaneously in a multi-threaded environment.

Besides renaming the clusters, CSMT also needs to rename the operands for any operation where an explicit cluster number is used. In our case, only the inter cluster communication operations send/recv use cluster numbers in their operands. The renaming of the operands can be done at any pipeline stage before execution of the operation, since the pipeline is tagged with a thread identifier and renaming is equivalent to adding the shift value to the cluster number. This can easily be done at the decode stage where the logic for identifying the operations is already available.

3.2. Thread Merge Hardware

Cluster assignment conflicts between threads are resolved on the basis of the individual priority of each thread. The execution packet is formed by merging instructions from as many threads as possible according to their priority. First, all the bundles from the highest priority thread are selected; then, bundles from the next priority thread are selected to be merged in execution packet if they do not collide with the already formed packet, and so on. Each cycle, a different priority is assigned to each thread in a round robin way. It is nevertheless possible to have different priority schemes which can be exposed to and controlled by the OS. For instance, a fixed priority scheme can be used for real-time applications, with the thread with real-time deadlines running with highest priority.

Figure 5 shows the thread merge hardware required to implement CSMT in a 4-thread 4-cluster architecture. This hardware consists of two parts: *Thread Select Logic* (TSL) and *Bundle Select Logic* (BSL). BSL selects a bundle from different threads on a per-cluster basis. TSL controls this selection. Assuming the same scenario of cycle 0 in Figure 3, TSL detects that Thread 1 cannot be scheduled, as there is a collision with the higher priority Thread 0 at physical cluster 1, but instructions from Threads 2 and 3 do not collide. So, TSL generates appropriate signals for BSL to merge threads 0, 2 and 3 and BSL selects bundles LC0 and LC1 from Thread 0, LC0 from Thread 2 and LC0 from Thread 3 (dark shaded), while bundles from Thread 1 are not selected for merging (light shaded). This logic is very simple, and for a 4-thread 4-cluster architecture requires approximately 300 transistors and is within 3 levels of gates delay. The implementation details can be found in [7] and are omitted from this paper due to space considerations.

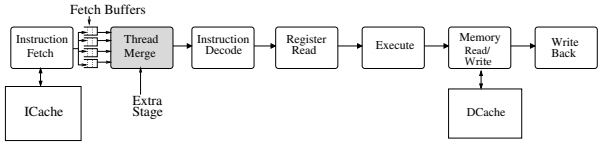

Figure 6: Processor pipeline

To prevent any negative effect on the cycle time, we assume that this hardware is in a separate pipeline stage. However, depending on the target frequency of the processor, it may be implemented in the instruction decode stage. The pipeline we use is similar to Lx pipeline except for an extra pipeline stage for thread merge, as shown in Figure 6. This increases the taken branch penalty to 2-cycles, which is 1 more than Lx[5].

4. Performance Evaluation

In order to test the efficacy of CSMT, experiments have been done in a 16-issue, 4-cluster architecture configuration (i.e. 4-issue per cluster). All the experiments have been done for a perfect memory model with no cache misses and for a real memory model (64KB, 4-way set-associative, 20-cycles miss penalty for both ICache and DCache) and assuming a target processor frequency of 250 MHz.

We have used a set of MediaBench [12] and relevant SpecInt 2000 [8] applications. We have also included production color space conversion [1] and imaging pipeline [21] benchmarks used in high performance printers. The benchmarks are shown in Table 1. Columns ILP_r and ILP_p show, for each benchmark, the ILP for real and perfect memory models respectively. Benchmarks are classified by their ILP in three categories: high ILP (colorspace and imgpipe), medium ILP (g721encode, g721decode, cjpeg and djpeg) and low ILP (mcf, bzip2, blowfish and gsmencode). This classification is shown in column $ILP\ Degree$ as L (low ILP), M (medium ILP) and H (high ILP).

The workloads used to evaluate CSMT are listed in Table 2. In order to select appropriate thread configurations, we have combined benchmarks with different ILP degrees, attempting to cover all possible combinations. Column la-

[5]In the Lx architecture, branch registers are read during instruction decode stage

Table 1: Benchmarks

Benchmarks	*ILP Degree*	*Description*	ILP_r	ILP_p
mcf	L	Minimum Cost Flow	0.96	1.34
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 2: Workload configurations

ILP Comb	*Thread 0*	*Thread 1*	*Thread 2*	*Thread 3*
LLLL	mcf	bzip2	blowfish	gsmencode
LMMH	bzip2	cjpeg	djpeg	imgpipe
MMMM	g721encode	g721decode	cjpeg	djpeg
LLMM	gsmencode	blowfish	g721encode	djpeg
LLMH	mcf	blowfish	cjpeg	colorspace
LLHH	mcf	blowfish	imgpipe	colorspace
LMHH	gsmencode	g721encode	imgpipe	colorspace
MMHH	djpeg	g721decode	imgpipe	colorspace
HHHH	imgpipe	colorspace	imgpipe	colorspace

beled as $ILP\ Comb$ indicates these ILP combinations. For example, configuration LLHH has two benchmarks with low ILP and two benchmarks with high ILP and configuration LMHH has one benchmark with low ILP, one benchmark with medium ILP and two benchmarks with high ILP.

We carried out the experiments by arranging the workloads in a multitasking environment as shown in figure 7. The number of threads supported by processor is exposed as virtual CPUs and the OS task scheduler schedules as many threads to run as the number of virtual CPUs with a timeslice of 1M cycles. After the expiry of the timeslice, a context switch takes place. The delay of a context switch is assumed to be neglegible. To improve fairness, after the context switch replacement threads are picked at random from the workload to alleviate any bias. For a single-thread processor, the threads run in serial order with a single thread running in the whole timeslice. For a 2-thread processor, 2 threads are scheduled to run together in the same timeslice and, for a 4-thread processor, 4 threads share the timeslice. The workloads are executed till one thread completes executing 100M VLIW instructions.

In order to compare CSMT to other techniques previously proposed in the literature, we have also evaluated the performance obtained by using a fine grained interleaved multithreading model (IMT) [19]. The architectural parameters are the same for IMT as for CSMT, except that the extra pipeline stage is not required in IMT and the taken branch penalty is 1 cycle instead of 2, as assumed in CSMT.

In figures 8 to 11, baseline is the original single-thread base architecture, while extrapipe is the single-thread base architecture with the extra pipeline stage to evaluate the impact of this extra stage. IMT2 and CSMT2 are 2-thread processor configurations for IMT and CSMT respectively,

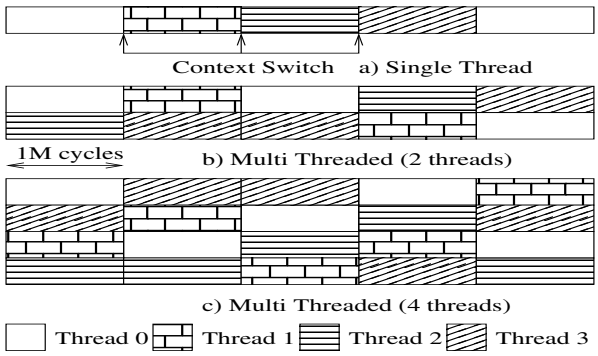

Figure 7: Multitasking execution for 4 threads

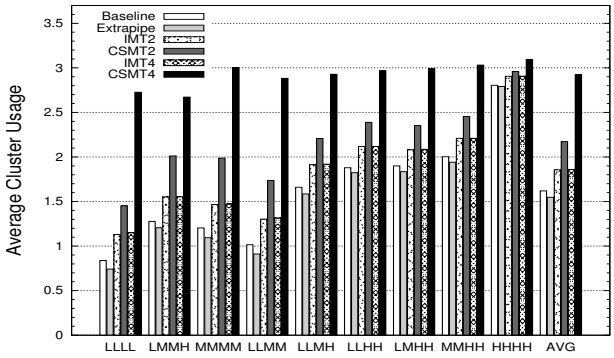

Figure 8: Cluster usage in perfect memory

and IMT4 and CSMT4 denote a 4-thread processor configuration (all configurations have 4 clusters with 4 issue width per cluster).

Figure 8 shows the cluster usage statistics for all the workload configurations assuming a perfect memory model with no cache misses. Cluster usage is the average number of clusters used per cycle by the workload during execution. It can range from 0 to the maximum value of 4. On a single thread processor (baseline and extrapipe), extrapipe has a little degradation in cluster usage because of the effect of the extra pipeline stage. However, the cluster usage improves considerably when the number of threads is increased, and more clusters are used most of the time. On average, the improvement of CSMT over IMT on a 2-thread processor is moderate (16.9%) because of the limited opportunities to merge threads. However, when a 4-thread processor is used, the average improvement in cluster usage is quite significant (57.9%), with an average cluster usage close to 3. Note that, in a high ILP workload like HHHH, there is little improvement in cluster usage over IMT. This is because very few opportunities to combine instructions exist, since both approaches already use a high number of clusters every cycle. However, in low ILP workloads like LLLL, where many opportunities exist to combine instructions, CSMT has a significantly higher cluster usage (28.5% for 2-threads and 81.7% for 4-threads).

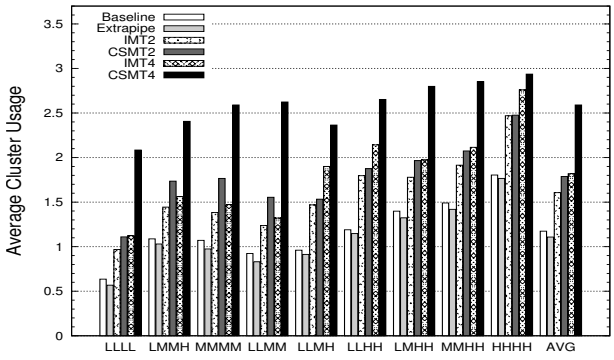

Figure 9: Cluster usage in real memory

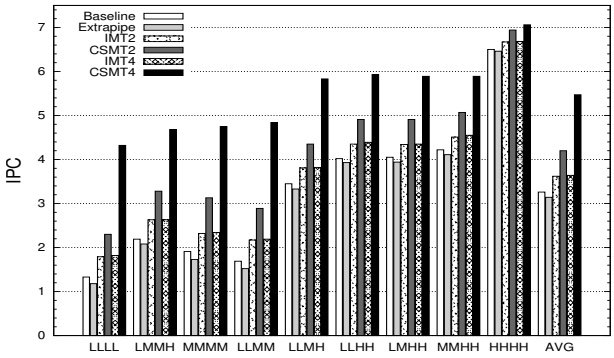

Figure 10: IPC in perfect memory

Figure 9 shows the cluster usage statistics when a real memory model is used. Notice that, with real memory, the improvement over IMT is not so significant for a 2-thread processor, though CSMT still does better (11.2% improvement). This is because, now, most of the time the inter-thread parallelism is used to hide cache miss stall cycles. However, with a 4-thread processor, CSMT does significantly better than IMT (42.3% on average), and even for the workload HHHH there is an improvement of 6.4%.

Finally, we have computed the IPC achieved by CSMT and IMT. Figure 10 shows the results obtained assuming a perfect memory model, and Figure 11 shows the same results with the real memory model.

The first thing to notice is that, with a perfect memory model (Figure 10), IMT does slightly better than the baseline processor. This is because, despite the fact that there are no vertical no-ops[6] 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. Our IMT implementation also hides these vertical no-ops by issuing instructions from an alternate thread. Since CSMT hides horizontal nops as well, it clearly outperforms IMT, specially with a a 4-thread processor configuration. In this case, CSMT (CSMT4) has an average speedup of 68% over the baseline and 50% over a 4-thread IMT configuration.

[6]A cycle where no instruction is issued

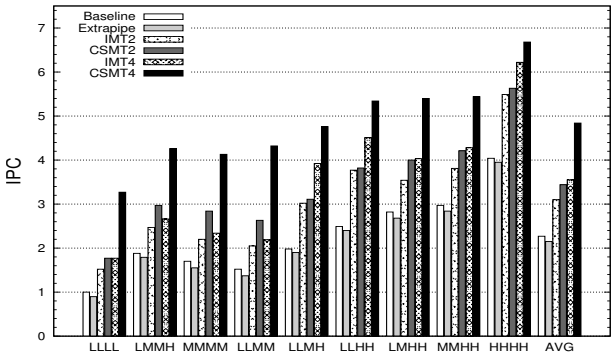

Figure 11: IPC in real memory

Notice that, for low ILP workloads like LLLL, the speedups are as high as 225% over the baseline and 137% over the 4-thread IMT configuration. Also notice that IMT with a 4-thread configuration achieves almost the same performance as with a 2-thread IMT because few vertical no-ops exist. CSMT with a 4-thread configuration, however, experiences a significant performance improvement (30.2%) over a 2-thread configuration (CSMT2), since there are more opportunities to fill up the horizontal nops. Even with the workload HHHH, CSMT with a 4-thread configuration has a visible improvement in IPC (1.7% over a 2-thread CSMT and 5.7% over a 4-thread IMT). Moreover, even a 2-thread CSMT outperforms 4-thread IMT by 17.4%. This shows the ability of CSMT to remove a significant part of horizontal waste.

Finally, when real memory is considered (Figure 11), the performance of a single-thread VLIW degrades significantly, while IMT and CSMT suffer only a minor impact. This fact shows the ability of both approaches to hide vertical no-ops. CSMT, however, still outperforms IMT by a significant margin. On average, a 2-thread CSMT configuration performs almost as well as a 4-thread IMT (IMT4 is only 3% better), while a 4-thread CSMT configuration outperforms both by a significant margin. For instance, a 4-thread CSMT configuration has an average speedup of 113% over the baseline, 41% over a 2-thread CSMT (CSMT2) and 36% over a 4-thread IMT configuration (IMT4). In particular cases, speedup with a 4-thread CSMT configuration can be as high as 227% over baseline and 97% over a 4-thread IMT configuration (LLMM). For the workload HHHH, a 2-thread CSMT configuration has little improvement of 2.5% over a 2-thread IMT configuration, but a 4-thread CSMT configuration has a noticeable improvement of 7.3% over a 4-thread IMT configuration.

Notice that, for single-thread configurations (baseline and extrapipe), there is a small performance degradation when an extra pipeline stage is assumed. This degradation will be noticeable in CSMT architectures when executing a single thread. However, that will happen only if the extra pipeline stage is actually required to meet cycle time

constraints. If the extra pipeline stage is not required, the CSMT performance for multithreaded configurations will be better than the results shown in this paper.

5. Conclusions

In this paper we have presented CSMT, a new approach to achieve the benefits of Simultaneous MultiThreading on clustered VLIW processors at a very small hardware cost. CSMT considers the set of operations that execute simultaneously in a given cluster (named bundle) as the assignment unit. All bundles belonging to a VLIW instruction from a given thread are issued simultaneously.

The analysis performed on a set of benchmarks using the Lx architecture [5] shows that, in general, no cluster is used during a significant amount of time due (mostly) to cache misses. Moreover, low ILP applications use only a few clusters most of the time. The compiler assigns operations mainly to the first clusters and tries to reduce the number of clusters used in order to reduce communication overhead among different clusters. As a consequence, the assignment of clusters collides when several threads are simultaneously executed. CSMT avoids this problem and allows a more parallel execution of the threads by renaming, at execution time, the clusters previously assigned by the compiler. The renaming mechanism is fast and has a very low hardware complexity (approx 300 transistors and 3 levels of gate delay).

CSMT implementation may require an extra pipeline stage if the renaming hardware cannot fit in the decode stage to meet the desired target frequency. However, the performance loss because of the extra pipeline stage in a single thread environment is very small (less than 3%).

Our results show that CSMT makes a better use of clusters than interleaved multithreading (IMT) with a negligible hardware cost. In terms of performance, CSMT significantly outperforms IMT. In general, CSMT for a 2-thread processor, achieves almost the same performance as IMT for a 4-thread processor and also outperforms it in some cases. For a 4-cluster machine with 4 threads, CSMT shows an average speedup of 68% over a single thread machine and of 50% over IMT assuming no cache misses, which shows the ability of CSMT to remove horizontal waste. When a realistic memory system is considered, the speedup of CSMT over single thread increases to 113% while speedup over IMT gets limited to 36%. This is because most of the resources wasted are due to stalls caused by cache misses, and IMT already does a good job hiding memory latency. However, CSMT still has a very significant advantage over IMT due to its ability to remove both vertical and horizontal waste.

References

[1] Colorspace Conversion Program Used in High Performance Printers, Personal Communication.

[2] D. Barretta, W. Fornaciari, M. Sami, and D. Bagni. Multithreaded Extension to Multicluster VLIW Processors for Embedded Applications. In *DATE*, pages 748–749, 2005.

[3] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. In *ASPLOS-II*, pages 180–192, 1987.

[4] J. R. Ellis. *Bulldog: a compiler for VLSI architectures*. MIT Press, Cambridge, MA, USA, 1986.

[5] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: A Technology Platform for Customizable VLIW Embedded Processing. In *ISCA*, 2000.

[6] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. Computers*, 30(7):478–490, 1981.

[7] M. Gupta, F. Sánchez, and J. Llosa. Merge logic for clustered multithreaded vliw processors. In *EUROMICRO Conference on Digital System Design*, 2007.

[8] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *IEEE Computer*, 33(7):28–35, 2000.

[9] F. Homewood and P. Faraboschi. ST200: A VLIW Architecture for Media-Oriented Applications. *Microprocessor Forum*, 2000.

[10] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the IA-64 Architecture. *IEEE Micro*, 20(5):12–23, 2000.

[11] B. Iyer, S. Srinivasan, and B. L. Jacob. Extended Split-Issue: Enabling Flexibility in the Hardware Implementation of NUAL VLIW DSPs. In *ISCA*, pages 364–375, 2004.

[12] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *MICRO*, 1997.

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

[14] E. Ozer and T. Conte. High-performance and low-cost dual-thread VLIW processor using Weld architecture paradigm. *IEEE Transactions on Parallel and Distributed Systems*, 16(12):1132–1142, 2005.

[15] B. R. Rau. Dynamically Scheduled VLIW Processors. In *MICRO*, pages 80–92, 1993.

[16] B. R. Rau, D. W. L. Yen, W. C. Yen, and R. A. Towle. The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs. *IEEE Computer*, 22(1):12–35, 1989.

[17] S. Rixner, W. J. Dally, B. Khailany, P. R. Mattson, U. J. Kapasi, and J. D. Owens. Register Organization for Media Processing. In *HPCA*, pages 375–386, 2000.

[18] N. Seshan. High VelociTI Processing. *IEEE Signal Processing Magazine*, 15(2):86–101, March 1998.

[19] B. J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. In *SPIE*, 1981.

[20] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *ISCA*, pages 392–403, 1995.

[21] VEX Toolchain. http://www.hpl.hp.com/downloads/vex/.

[22] W.-D. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: preliminary results. In *ISCA*, pages 273–280, 1989.