Performance Evaluation of CSMT 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 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. CSMT (Cluster-level Simultaneous MultiThreading) allows some degree of SMT in clustered VLIW processors. 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. Also 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.

KEYWORDS: Clustering, VLIW processor, Multithreading, CSMT

1 Introduction

In VLIW architectures, exploiting high amounts of ILP is a difficult task because the issue width is limited by the number of functional units (FUs) that are connected to a single register file. Register file access time, area, and the complexity of the bypassing network grow with the number of FUs. Clustered VLIW architectures tackle this problem by introducing more than one register file and clustering the functional units according to the register files they are connected to. This approach allows higher levels of issue width than unicluster VLIW architectures.

However, the ILP exposed in most of the applications is limited and the processor is heavily under utilized. Simultaneous MultiThreading (SMT) [1] is a well known technique to improve the resource utilization by exploiting thread level ILP. However, implementing SMT is not feasible for embedded VLIW processors because of the extra complexity of dynamic resource allocation and conflict management. In clustered VLIW, static resource partitioning at cluster level can be used for a low complexity multithreading. Our proposal [2, 3], is similar to Simultaneous MultiThreading but the granularity is at cluster level instead of operation level.

If we analyze the cluster usage of most programs, there is a big load bias on the cluster 0 compared to other clusters and, on increasing the issue width per cluster, the bias increases

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

PC0	PC1	PC2	PC3
A_0	B_0	–	–
–	B_1	–	–

Thread 0

PC0	PC1	PC2	PC3
–	A_0	B_0	–
–	–	B_1	C_1

Thread 1

PC0	PC1	PC2	PC3
–	–	A_0	–
C_1	D_1	–	B_1

Thread 2

PC0	PC1	PC2	PC3
–	–	–	A_0
B_1	–	–	A_1

Thread 3

(a) 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_1	B_1	C_1

(b) Merged instruction stream

Figure 1: Instruction stream on a 4-thread 4-cluster architecture

further. This is quite reasonable, since the compiler tries to schedule most of the operations in the same cluster to avoid communication overhead. Moreover, only a small number of clusters are actually used most of the time since there is not always enough parallelism available during a program's execution.

If we try to implement multithreading in a naive way, most of the threads would compete among resources on a few clusters most of the time rather than using resources in other clusters which are not used at all. If we can schedule other threads so that they use different clusters, instead of the ones assigned by the compiler, then higher ILP would be achieved.

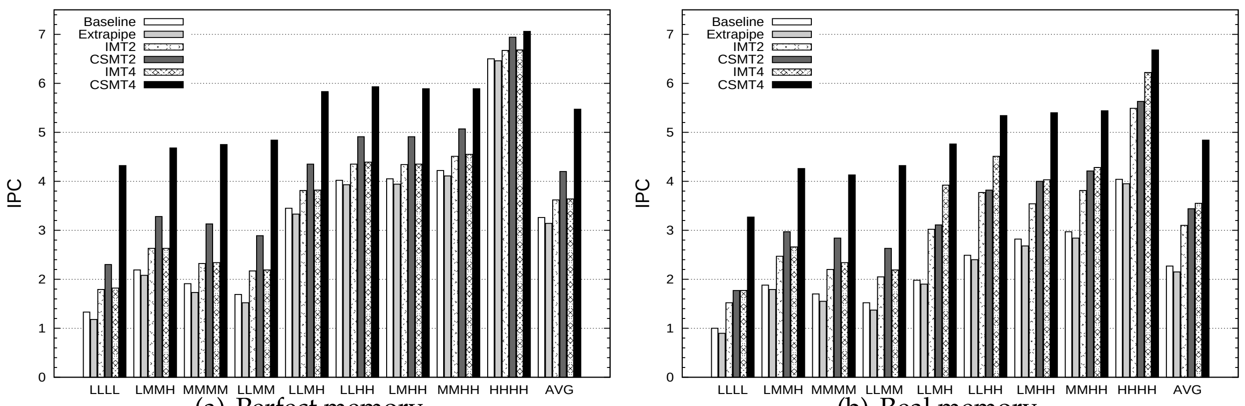
2 Clustered SMT Architecture

CSMT architecture [2] is based on the VEX clustered architecture which is modeled upon the HP/ST Lx [4] VLIW family. The VEX C compiler used in this study is a derivation of HP/ST ST220 C compiler, that uses *Trace Scheduling* [5] as global scheduling algorithm.

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. 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. 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.

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. The merging mechanism is fast and has a very low hardware complexity (approx 300 transistors and 3 levels of gate delay for a 4-thread 4-cluster architecture [6]). CSMT implementation may require an extra pipeline stage if the renaming hardware cannot fit in the decode stage to meet the desired target frequency.

Figure 1 shows a sample multithreaded execution for a 4-thread 4-cluster architecture using CSMT. Figure 1(a) shows the physical mapping of clusters (PC0-PC3) done by CSMT after cluster renaming for each thread at execution time. A_0 means the group of operations



(a) Perfect memory (b) Real memory

Figure 2: Performance Results

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.

The effect of using CSMT is shown in Figure 1(b). The sequential execution of the four threads in a machine with perfect memory would require 8 cycles. CSMT, however, would require only 4 cycles. 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. 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.

3 Results

We have used a set of MediaBench and relevant SpecInt 2000 applications. We have also included production color space conversion and imaging pipeline benchmarks used in high performance printers. Benchmarks are classified by their ILP in three categories: low ILP (L: mcf, bzip2, blowfish and gsmencode, IPC < 1.5), medium ILP (M: g721encode, g721decode, cjpeg and djpeg, IPC < 4) and high ILP (H: colorspace and imgpipe, IPC > 4). In order to select appropriate thread configurations, we have combined benchmarks with different ILP degrees, attempting to cover all possible combinations. For example, configuration LMHH has one benchmark with low ILP, one benchmark with medium ILP and two benchmarks with high ILP. 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.

We also evaluated the performance obtained by using a fine grained interleaved multithreading model (IMT) [7]. The architectural parameters are the same for IMT as for CSMT, except that the extra pipeline stage is not required in IMT. In figures 2(a) and 2(b), 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, and IMT4 and CSMT4 denote a 4-thread processor configuration (all configurations have 4 clusters with 4 issue width per cluster).

For perfect memory, CSMT (CSMT4) has an average speedup of 68% over the baseline and 50% over a 4-thread IMT configuration. 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. 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.

When real memory is considered (64KB, 4-way set-associative, 20-cycles miss penalty for both ICache and DCache), 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).

4 Conclusions

In this paper we have presented the performance evaluation of CSMT, a new approach to achieve the benefits of Simultaneous MultiThreading on clustered VLIW processors at a very small hardware cost. CSMT 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).

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, and when a realistic memory system is considered, CSMT has a speedup of 113% over single thread and 36% over IMT.

References

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

[2] Manoj Gupta, Josep Llosa, and Fermín Sánchez. Performance Evaluation of CSMT for VLIW Processors. Technical Report UPC-DAC-RR-2007-23.

[3] Manoj Gupta, Josep Llosa, and Fermín Sánchez. Cluster Level Multithreading for VLIW Processors. In *ACACES*, 2006.

[4] 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.

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

[6] Manoj Gupta, Fermín Sánchez and Josep Llosa. Merge Logic for Clustered Multithreaded VLIW Processors, To be published in EUROMICRO Conference on Digital System Design. 2007.

[7] B. J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. In *SPIE*, pages 241–248, 1981.