A Low Cost Split-Issue Technique to Improve Performance of SMT Clustered VLIW Processors

Manoj Gupta **Fermín Sánchez** **Josep Llosa**

Department of Computer Architecture
Universitat Politècnica de Catalunya
Barcelona, Spain
{*mgupta,fermin,josepll*}*@ac.upc.edu*

***Abstract*—Very Long Instruction Word (VLIW) processors are a popular choice in embedded domain due to their hardware simplicity, low cost and low power consumption. Simultaneous MultiThreading (SMT) is a popular technique for improving processor performance. To maintain execution semantics, a VLIW instruction needs to be issued in entirety, which restricts the opportunities in SMT. Split-issue at operation-level is a technique that allows issuing a VLIW instruction in parts without breaking execution semantics. Issuing an instruction in parts allows non-conflicting part of an instruction to be issued along with other instructions and improves SMT performance. However, implementing split-issue at operation-level requires complex structures and is not practical for an embedded VLIW processor. This paper proposes cluster-level split-issue, which implements split-issue at a cluster-level boundary for clustered VLIW processors. Cluster-level split-issue has a very low hardware overhead in contrast to split-issue at operation-level. Experimental results show that cluster-level split-issue, despite being more restrictive than split-issue at operation-level, achieves similar performance and improves SMT performance significantly.**

***Keywords*-Clustered VLIW Processors; Multithreading;**

I. Introduction

Very Long Instruction Word (VLIW) processors have wide acceptance in the embedded domain due to hardware simplicity, low cost and low power consumption [1], [2], [3]. To exploit high Instruction Level Parallelism (ILP), VLIWs need to be designed with a significant issue width. However, the centralized Register File (RF), with all the Functional Units (FUs) connected to it, becomes a bottleneck because of an increase in RF delay, power consumption and area [4]. Clustered VLIW architectures have multiple RFs and cluster the FUs according to the RFs they are connected to. Many VLIWs have been designed using the clustered approach [2], [3].

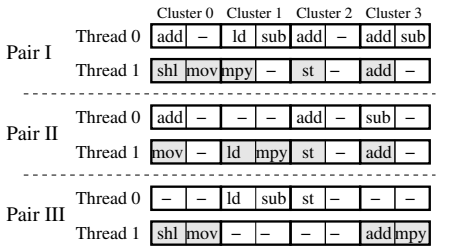
Some applications do not take advantage of the high issue width available in a VLIW processor and the processor is heavily underutilized. In the context of VLIW architectures, processor underutilization can be characterized in terms of vertical and horizontal waste. Vertical waste are the cycles where no operations are issued at all. Horizontal waste is the underutilization of the issue width of the processor, i.e. the number of operations issued in a cycle is less than the issue width. Several multithreading techniques have been proposed to reduce the vertical and horizontal waste in the processor. **Block MultiThreading** (BMT) [5], [6] executes instructions from a single thread until it is blocked by a long latency event (e.g. a cache miss) and then starts the execution of a new thread. **Interleaved MultiThreading** (IMT) [7], [8] does a zero cycle context switch every cycle, so that instructions from different threads are interleaved at execution time. Both BMT and IMT reduce only the vertical waste.

Simultaneous MultiThreading (SMT) [9] is a popular technique that reduces both horizontal and vertical waste in the processor. SMT issues multiple instructions from multiple threads each cycle. In a SMT processor, issue-slots of the processor are filled by operations of different threads, converting thread level parallelism (TLP) into ILP, thus improving processor performance.

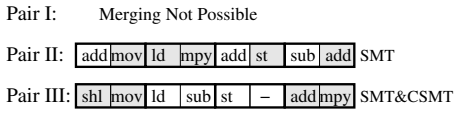
Cluster-level Simultaneous MultiThreading (CSMT) [10] is a variant of SMT specifically proposed for clustered VLIW processors. In CSMT, the instruction merging is done at a cluster-level granularity instead of the fine-grain merging at operation-level done by SMT. Hence, CSMT issues simultaneously instructions from multiple threads only when the threads use different clusters. This restricts the opportunities to merge instructions in comparison to SMT, but allows for a lower complexity implementation of the merging hardware.

To illustrate how instructions from different threads are merged in CSMT, Figure 1(a) displays 3 pairs of instructions for 2 threads for a 4-cluster 2-issue per cluster (8-issue) architecture. In the figure, operations in the white background belong to Thread 0 and operations with a grey background belong to Thread 1. Note that, if a pair of instructions can be merged by CSMT, it can always be merged by SMT but not vice-versa. Also note that if both CSMT and SMT can merge a pair of instructions, the final merged instruction is identical for both SMT and CSMT. The final instructions obtained by merging are shown in Figure 1(b). Neither CSMT nor SMT can merge Pair I because of conflicts at clusters 0, 1 and 3, both at operation-level and cluster-level.

978-1-4244-6443-2/10/$26.00



(a) Pairs of instructions to merge



(b) Merged instructions

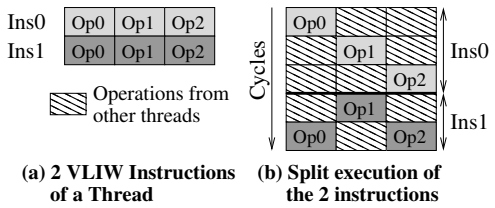
Figure 1: Instruction Merging in SMT and CSMT

Pair II can be merged by SMT since there are no conflicts at operation-level. CSMT, however, cannot merge this pair, since both instructions in the pair use clusters 0, 2 and 3. As CSMT checks resource conflicts at the cluster-level, there is a conflict at these clusters. Pair III, however, can be merged by CSMT (and SMT as well) as the first instruction uses only clusters 1 and 2 which are not used by the other instruction.

Note that the VLIW instructions are merged in their entirety for both CSMT and SMT. This happens because the compiler for a VLIW processor schedules instructions assuming that all the operations in the instruction are simultaneously executed. Hence, instructions need to be executed in their entirety to avoid breaking execution semantics. As a result of this constrain, the opportunities to reduce horizontal waste are restricted. For instance, for the pair I in Figure 1(a), only one of the two instructions can be selected at a time and some issue-slots are wasted.

Split-issue [11], [12] removes the constrain of having to issue all operations in the instruction simultaneously while honoring execution semantics. Split-issue does a dynamic scheduling of the operations of a VLIW instruction. Thus, the operations of a VLIW instruction can be flexibly issued. The flexibility in issuing operations of an instruction allows parts of an instruction from one thread to be executed along with an instruction of another thread. Hence, a further reduction in the horizontal waste is achieved. For instance, the two empty issue-slots of the instruction of Thread 0 of Pair I in Figure 1 can be filled by selecting two operations from instruction of Thread 1. However, the hardware overhead because of the dynamic scheduling requirement is not trivial and takes away the low power and low cost advantages of VLIW processors.

In this paper, we propose a variant of split-issue for clustered VLIW processors which we refer to as cluster-level split-issue. In cluster-level split-issue, the instruction can be split at a cluster boundary i.e. all the individual



(a) 2 VLIW Instructions of a Thread **(b) Split execution of the 2 instructions**

Figure 2: Instruction Execution with Split-Issue

operations belonging to a bundle[1] are issued as a unit. Bundles themselves can be issued in separate cycles. This avoids the complex hardware requirement as no dynamic scheduling of operations requires to be done. In clustered VLIW processors, no dependencies[2] exist between different bundles as they read from and write to different register files. Hence, issuing of the bundles in different cycles does not break execution semantics. Furthermore, performance obtained by use of cluster-level split-issue is very close to the performance obtained by prior split-issue proposals.

The rest of the paper is organized as follows. First, Section II discusses prior split-issue proposals. Cluster-level split-issue is discussed in detail in Section III. The base architecture used for the evaluation is presented in Section IV. Several issues that arise because of split-issue are discussed in Section V. A detailed performance evaluation is presented in Section VI. Finally, Section VII concludes the paper.

II. Split-Issue

Split-issue [11], [12] allows issuing of the operations of a VLIW instruction in parts instead of a complete unit. Split-issue was originally proposed as a mechanism to maintain binary compatibility in VLIW processors [11]. However, it can also be used to improve processor throughput in SMT [12] as parts of an instruction of one thread can be executed along with an instruction of another thread. Figure 2 shows an example of split execution for 2 instructions, Ins0 and Ins1, of a thread on a multithreaded 3-issue VLIW processor. Both instructions have 3 operations each, Op0-Op2, as shown in Figure 2(a). Without split-issue, all operations of an instruction would have to be issued at the same cycle. Using split-issue, however, removes this constrain and the operations can be flexibly issued in any order. Figure 2(b) shows a sample split execution of the operations of the 2 instructions. Operations of the first instruction, Ins0, are issued in 3 separate cycles and operations of the second instruction, Ins1, are issued in 2 cycles. Rest of the issue-slots are filled by operations from other threads. Note that operations from instruction Ins1 are not issued until all operations of the instruction Ins0 have been issued, i.e. the

[1]We use the same terminology as the Lx architecture. An operation is the basic execution unit; the operations scheduled to execute at a given cluster constitute a bundle and the set of bundles form the VLIW Instruction.

[2]Only exception is the inter-cluster communication operations.

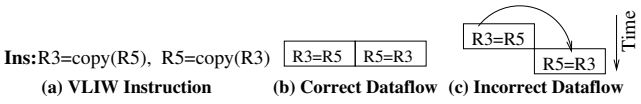


Figure 3: Issues with Dataflow

Split−type	Merging policy: operation−level (SMT)	Merging policy: cluster−level (CSMT)
operation−level	OOSI	−
cluster−level	COSI	CCSI

Figure 4: Applicability of split-issue

in-order[3] relationship between the VLIW instructions is still maintained.

A. Issues with split-issue

In general, the ISA and architecture are tightly coupled in VLIW processors and the actual latencies of the operations are exposed to the compiler or the programmer. The dependency check and latency-cognizant instruction scheduling is done entirely at compile time. Instructions are scheduled with the assumption that all operations belonging to an instruction are issued simultaneously. No runtime dependency checking or interlocking hardware is required. The compiler's data dependency assumptions are potentially violated if the operations of an instruction are not simultaneously issued. To illustrate this fact, Figure 3(a) shows a VLIW instruction consisting of 2 operations. The instruction does a single cycle swap of the registers R3 and R5 (of the same cluster) without using extra registers and it is a legal VLIW instruction. The two operations read old values of the source registers if the operations are not split, as shown in Figure 3(b). Now, let us assume that the 2nd operation is issued at a later cycle. The delayed issue results in an incorrect dataflow as shown in Figure 3(c), since the 2nd operation will read an incorrect value of register R3. Hence, measures are required to avoid breaking the compiler dataflow assumptions. Following section discusses the implementation of split-issue while maintaining the correct dataflow.

B. Implementation of split-issue

For split-issuing of the VLIW instructions, the operations of the VLIW instructions are first divided into 2 phases at runtime. Phase I of an operation performs the computation and writes the result to a delay buffer. Phase II copies the value from the delay buffer to the original destination register. After the division of the operations of a given VLIW instruction into the two phases, an amalgamated instruction corresponding to the given VLIW instruction is formed. For a given VLIW instruction, phase I of all the operations are part of the corresponding amalgamated instruction. Unlike phase I of an operation which always belong to the corresponding amalgamated instruction, phase II of an operation can belong to the corresponding amalgamated instruction itself or a later one, depending upon the FU latency. For a FU with a latency of N cycles, the phase II for that operation is part of the amalgamated instruction corresponding to the (N-1)th VLIW instruction later. Thus, an amalgamated instruction for a given VLIW instruction contains the phase I of all the operations of the given VLIW instruction, phase II of the operations of the corresponding VLIW instruction with unit latency, and phase II of the operations belonging to earlier VLIW instructions because of non-unit latencies. Once an amalgamated instruction is obtained, phase I of the operations belonging to the amalgamated instruction can be issued dynamically. All phase II in the amalgamated instruction are issued simultaneously with the last phase I operation of the instruction. Issue of phase II is integrated with the delay buffers implementation and does not consume issue-slots [12]. Note that phase I of the operations from the next amalgamated instruction are issued only when all phase I of the operations of the current amalgamated instruction have been issued. This enforces that the execution is still in-order wrt the VLIW instructions.

Split-issue allows a flexible and dynamic issuing of operations of a VLIW instruction creating more opportunities for removing horizontal waste. However, this flexibility comes at the cost of the extra hardware required to honor execution semantics. As the operations are dynamically issued, the issue logic has a complexity similar to the issue logic in superscalars. For instance, an issue queue logic of 32 entries is required for supporting split-issue on a 4-thread 8-issue VLIW processor. Also, a logic similar to register renaming is required for assigning the delay buffers. Both issue queue and register renaming are amongst the most complex and power hungry structures even on superscalar processors [13]. Addition of these structures takes away the key advantages of low power and low cost of VLIWs making them impractical[4] for use in embedded domain.

III. CLUSTER-LEVEL SPLIT-ISSUE

In this paper, we propose a restricted variant of split-issue for clustered VLIW processors which we refer to as cluster-level split-issue. In comparison to earlier proposals, where the operations of an instruction can be issued in any order, cluster-level split-issue allows instructions to be split

[3]In theory, operations from instruction Ins1 may be executed with the operations of Ins0 as long as the data dependencies are taken care of. However, this changes the VLIW processor to a full fledged out-of-order processor with a VLIW ISA, requiring complex and power hungry hardware. This complexity is not desirable for VLIWs, whose prime attraction is their low complexity & low power.

[4]Even the original split-issue proposal [11] is also not in favor of split-issue because of complex hardware requirements for split-issue.

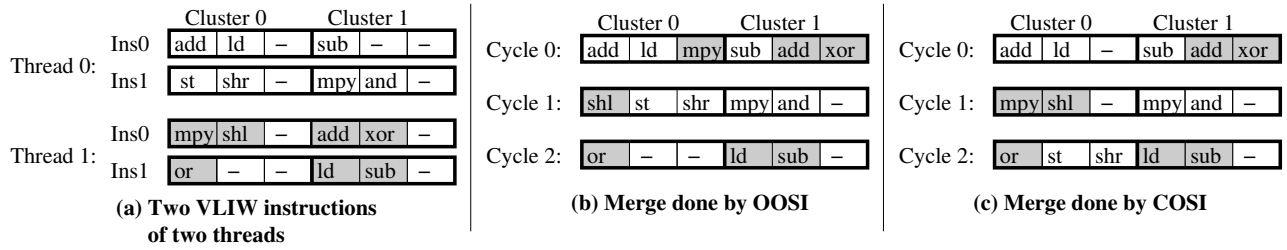


Figure 5: Operation-level and cluster-level split-issue with operation-level merging

only at a cluster-level boundary, i.e. splitting of operations belonging to the same bundle is not allowed. For the sake of clarity and to avoid confusion, we refer to the split-issue proposal detailed in Section II as operation-level split-issue from now onwards.

In a clustered VLIW processor, no dependencies exist between different bundles, as they read from and write to different register files. Hence, the bundles of an instruction can be independently issued without breaking execution semantics. Cluster-level split-issue is more restrictive than operation-level split-issue and operations inside a bundle are not allowed to be split. The split is at the cluster-level boundary i.e. on a bundle basis. This avoids a dynamic scheduling of the operations inside the bundle. Furthermore, operations do not have to be split into different phases, which is a primary requirement for operation-level split-issue. The bundles themselves are dynamically issued but the dynamic issuing is achieved with only small changes in the merging hardware and does not increase the hardware complexity (explained in detail in Section V).

Figure 4 shows the applicability of the two split-issue techniques (operation-level and cluster-level) for the two multithreading schemes, SMT and CSMT. We would like to remind that CSMT merges instructions at a cluster-level granularity, while SMT merges the instructions at an operation-level granularity. Enhancing instruction merging with split-issue techniques leads to the following configurations as shown in Figure 4.

OOSI: Operation-level split and operation-level merging (previously proposed split-issue technique).

COSI: Cluster-level split but operation-level merging.

CCSI: Cluster-level split and cluster-level merging.

As shown in the figure, cluster-level split-issue can be used with both operation-level and cluster-level merging. However, operation-level split-issue makes sense only with operation-level merging. To illustrate the difference between operation-level and cluster-level split-issue when operation-level instruction merging is used (OOSI and COSI), Figure 5 shows an example showing the merging done by both techniques. We assume that number of issue slots is the only critical resource in this example. Also, it is assumed that priorities for merging changes each cycle between the threads in a round-robin way. Thus Thread 0 has the higher priority in the first cycle, but in the second cycle, Thread 1 has the higher priority, and so on. Figure 5(a) shows 2 instructions each for 2 threads on a 2-cluster 6-issue architecture (3-issue per cluster). Instruction Ins0 from Thread 0 uses two issue slots in cluster 0 and one in cluster 1. While, instruction Ins0 from Thread 1 uses two issue slots in both cluster 0 and cluster 1. Without split-issue, both instructions cannot be merged because of insufficient number of issue slots in cluster 0. Hence, only instruction Ins0 of Thread 0 is issued at the first cycle. In the second cycle, Thread 1 has the higher priority. Again instruction Ins0 of Thread 1 cannot be merged with instruction Ins1 of Thread 0. This forces Ins0 of Thread 1 to be issued alone and so on. The execution would require 4 cycles to execute the instructions without split-issue, since merging of the instructions of the two threads is not possible at any cycle. Using split-issue, however, reduces the number of execution cycles from 4 to 3.

Figure 5(b) shows the execution of the instructions when operation-level split-issue (OOSI) is used. At cycle 0, mpy operation from cluster 0 of Thread 1 can be issued with the operations of cluster 0 of Thread 0. Similarly, at cluster 1, operations add and xor operations of instruction Ins0 of Thread 1 are issued at cluster 1 along with sub operation of instruction Ins0 of Thread 1. At cycle 1, operations st and shr of instruction Ins1 of Thread 0 can be issued with the remaining operations of instruction Ins0 of Thread 1. The remaining operations of Thread 1 (i.e. Ins1) are issued at the third cycle.

The execution of the instructions of two threads when cluster-level split-issue (COSI) is used is shown in Figure 5(c). With COSI, mpy and shl operations of instruction Ins0 of Thread 1 cannot be issued at different cycles. Hence, no operations are selected from cluster 0 of Thread 1 at cycle 0. However, operations add and xor of cluster 1 of Thread 1 are issued with sub operation of cluster 1 of Thread 0 because no splitting of operations inside the bundle is required. At cycle 1, Thread 1 has the higher priority. Thus, the remaining operations of instruction Ins0 of Thread 1 (mpy and shl of cluster 0) are issued. Operations of cluster 1 from instruction Ins1 of Thread 0 are also issued at cycle 1. At cycle 2, pending operations of instruction Ins1 of Thread 0 are issued and are merged with instruction Ins1 of Thread 1.

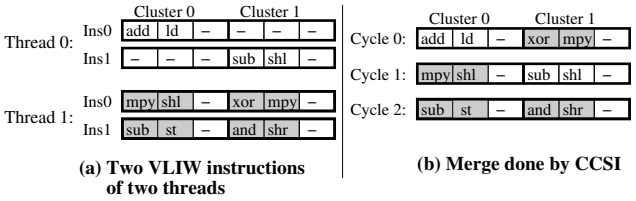


Figure 6: Cluster-level split-issue with cluster-level merging

Note that even though execution using OOSI or COSI takes the same number of cycles in the example, OOSI is more efficient than COSI. For instance, COSI issues operations from both Thread 0 and Thread 1 at cycle 2. OOSI, however, issue operations only from Thread 1. The operations of Thread 1 can be merged with the next instruction (Ins2) of Thread 0 further improving OOSI performance.

Next we present an example of cluster-level split-issue with cluster-level merging (CCSI). Figure 6(a) shows 2 VLIW instructions each for 2 threads. Instruction Ins0 of Thread 0 uses only cluster 0 but instruction Ins0 belonging to Thread 1 uses both clusters. Hence, the two instructions cannot be simultaneously issued in as both of them use cluster 0. The same priority rotation policy used in the previous example is assumed for the threads. Without split-issue, normal execution would require 4 cycles as no merging is possible at any cycle. Using CCSI, however, reduces the execution time to 3 cycles by creating more opportunities for merging. Figure 6(b) shows the execution of the two threads when CCSI is used. At cycle 0, operations belonging to cluster 1 of Thread 1 can be issued with Thread 0, as only cluster 0 is used by instruction Ins0 of Thread 0. In the second cycle, operations from cluster 0 of instruction Ins0 of Thread 1 are issued. Operations from cluster 1 of instruction Ins1 of Thread 0 are also issued as cluster 1 is no longer used by Thread 1. Finally, at cycle 2, instruction Ins1 of Thread 1 is issued. As illustrated in the examples, shown in figures 5 and 6, using split-issue further improve processor performance in a multithreaded environment. The extra performance is achieved because more opportunities for merging instructions are created.

IV. Base Architecture

The evaluation done in this paper is based on the VEX clustered architecture [14] modeled upon the commercial HP/ST ST200 [2], [1] VLIW family. The VEX C compiler [14] used in this study is a derivation of the HP/ST ST200 C compiler, which itself is a derivative of the Multiflow compiler [15] that uses *Trace Scheduling* [16] as global scheduling algorithm and *Bottom Up Greedy* [17] 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* machine i.e. the actual latency of any FU can be shorter than the compiler assumption. No stalls or interlocks are required if hardware can complete an operation in the same or fewer cycles. However, for operations like memory accesses, which may take longer than the assumed latency, execution is stalled until the architectural assumptions hold true. For our evaluations, a 4-issue per cluster configuration is assumed. A 4-issue cluster has 2 multipliers, 1 load/store unit, and 4 ALUs. Memory and multiply operations have a 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. A fully connected inter-cluster interconnection network is assumed between the clusters.

To improve multithreading performance, Cluster Renaming [10] is used in all our experiments for both SMT and CSMT. Cluster renaming reduces the bias on heavily used clusters by statically distributing the clusters of each thread. The renaming mechanism performs a rotation of the the original cluster assignment done by compiler by a given renaming value. The renaming value of each thread is a fixed number computed at design time, based on the number of clusters and the number of simultaneous threads supported by the processor. For instance, in a 4-thread 4-cluster machine, Thread 0 is rotated by 0, Thread 1 by 1, Thread 2 by 2, and Thread 3 by 3.

V. Cluster-level Split-Issue Implementation and Issues

This section describes the major hardware changes required to implement cluster-level split-issue. We also discuss several issues that arise because of using split-issue. In particular, we discuss issues like supporting precise exceptions, issues of register file and memory port contentions and handling of inter-cluster communications. Note that all these cases related to inter-cluster communications, precise exceptions etc. are not particular to cluster-level split-issue but are equally applicable to operation-level split-issue as well.

A. Merging hardware

This section describes the implentation of the original SMT/CSMT merging hardware and the changes required in it to support cluster-level split-issue. First, Figure 7(a)

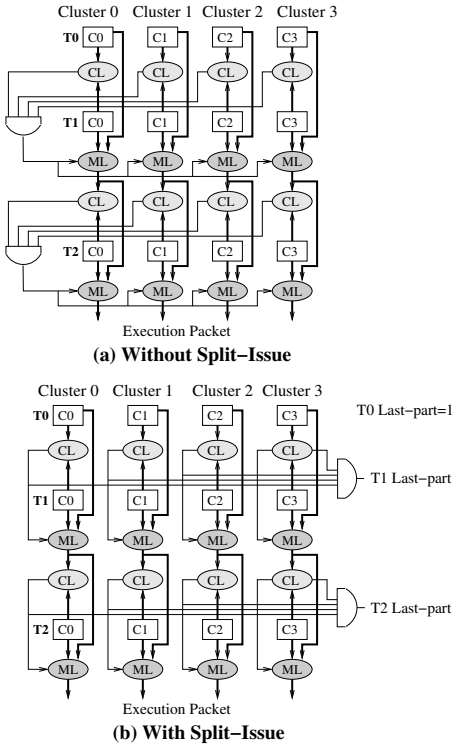


Figure 7: Merging Hardware

shows the original merging hardware for a 4-cluster 3-thread architecture. In the figure, T0-T2 represent the 3 threads, C*i* represents the operations of the bundle assigned to cluster *i* for a given thread, CL is the collision detection logic and ML is the merge logic. CL checks if there is a resource conflict between the two bundles given as input. ML merges the bundles of two instructions corresponding to the same cluster. The output of ML is controlled by a signal which dictates whether the output is a merged bundle of the inputs or the first input is passed as the output (i.e. no merging is possible). For merging two instructions, the merging hardware has to check for resource conflicts at all the clusters. Only when there are no resource conflicts at all the clusters (enforced by the AND gates), two instructions are merged. The internal implementation of CL and ML varies depending on the approach used for merging (operation-level or cluster-level). We assume that Thread T0 has the highest priority for merging, Thread T1 has the next level of priority and Thread T2 has the lowest priority (computation of priority is independent of the merging hardware implementation). Thus, first threads T0 and T1 are tried for merging. Then, the output of this merge (T0 merged with T1 if merging was possible or only T0 if T0 and T1 could not be merged) is tried to be merged with Thread T2. The output of the final merge forms the execution packet.

The merging hardware required for supporting cluster-level split-issue requires only minor modifications in the

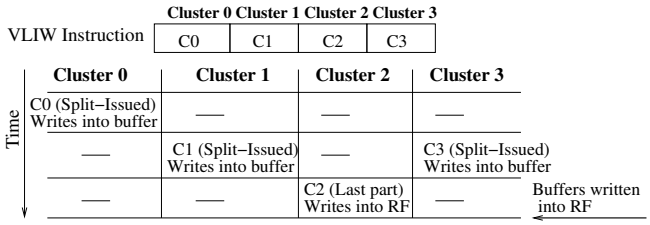


Figure 8: Delaying updates to architectural state by using buffers

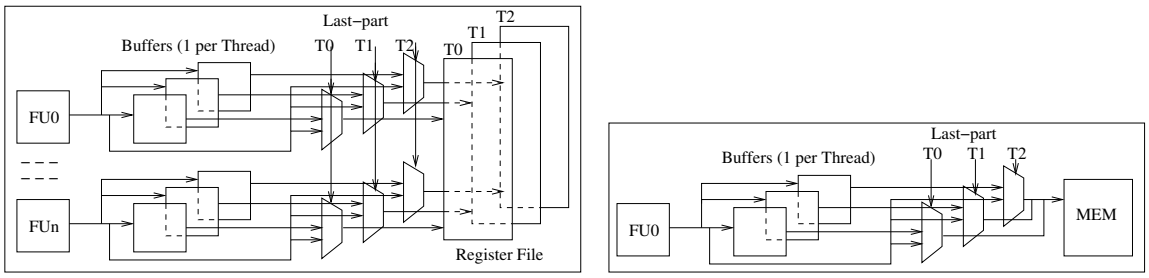
original SMT/CSMT merging hardware as shown in Figure 7(b). Without cluster-level split-issue, two instructions can be merged only when there are no conflicts at any of the clusters. On the other hand, when cluster-level split-issue is supported, the resource conflict status of other clusters is not required to merge the bundles corresponding to a given cluster. Hence, the merging process is completely independent for each cluster and, in fact, results in a lower delay for the merging hardware.

The merging hardware with cluster-level split-issue support also generates a last-part signal for each thread. Last-part signal indicates whether the instruction of a given thread has been merged in its entirety or not. The generation of this signal is not on the critical path of the merging hardware and does not affect the delay of the merging hardware. This signal is labeled as T*i* Last-part for Thread T*i* as shown in Figure 7(b). Last-part signal is required at a later pipeline stage (explained in the following section). Note that Thread T0 is always selected in its entirety because it is the highest priority thread.

B. Issues with Exceptions/Interrupts

Using split-issue may result into an inconsistent architectural state which has a direct impact on supporting precise exceptions/interrupts. Restoration to a consistent state is mandatory before an exception/interrupt is taken. However, if split-issue is used, restoration to a consistent state may not be possible. For instance, let us assume that an instruction is issued in two parts. The first part executes without raising any exception and updates the architectural state by writing into register file or memory. The second part, however, raises an exception. To take the exception, the architectural state should be rolled back to the state of the VLIW instruction just before the excepting instruction. However, the rollback is not possible because of the updates already made by the first part. Hence, the split-issued operations should not update the architectural state of the processor to allow restoration to a consistent state. For doing so, buffers are required to hold values that the split-issued operations write into register file and memory.

Figure 8 illustrates the usage of buffers on a 4-cluster architecture. In the figure, bundles C0-C3 refer to the groups of operations scheduled to execute in clusters 0-3 respec-



(a) Buffer organization for register file **(b) Buffer organization for memory**

Figure 9: Buffer organization

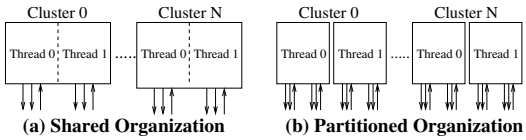
tively. If the operation executed by a FU is split-issued, then the result is written to the buffer. If the operation is not split-issued (i.e. the operation belongs to the last part of instruction), the result is directly written to register file (or memory). The results from the buffers are written to register file (or memory) when the last part of the instruction is executed. Bundle C0 (i.e. all operations belonging to cluster 0) is issued first, bundles C1 and C3 are issued next and finally bundle C2 is issued. Bundles C0, C1 and C3 are split-issued and write into buffers. However, bundle C2, being the last part, writes directly into its RF. The content of the buffers holding the results of bundles C0, C1 and C3 are also written to their corresponding register files at the same time as C2.

Since the execution is always in-order between VLIW instructions, results from only one VLIW instruction per thread may have to be stored in buffers at any time. Thus, the storage requirement for the buffers to hold the results for each thread that are going to be written into RF is the same as the issue-width of the processor. Figure 9(a) shows the organization of the buffers used for register file to hold the values temporarily till the last part of an instruction is executed The results (from FU or from buffers) are written to RF only when the Last-part signal (already computed at issue stage by merging hardware, Figure 7(b)) is set.

An extra set of buffers is required to hold the memory writes, which is equal to the number of memory functional units. Figure 9(b) shows the organization of buffers for memory. The buffers used to store the split-issued results both for memory and RF neither require to be multiported nor any data forwarding is required. Hence, little hardware overhead is incurred because of the buffers. Delaying the updates to the architectural state, however, creates another issue because of a contention for register file and memory ports. These issues are discussed in the forthcoming sections.

C. Register file port contention

It is possible that last part of the instructions from multiple threads may be executed at the same time. For instance, the instruction shown in Figure 8 updates the values from buffers to register file and memory only when its last part is executed



(a) Shared Organization **(b) Partitioned Organization**

Figure 10: Register File Organizations

at cluster 2. There is a possibility that another thread might have its last part executing at the same time. Hence, the operations being executed may contain the last parts of the instructions of several threads. Now, results of the operations of several threads need to be written to the register file at the same time. To allow all these writes to register file, W register file write ports per thread must be available at each cluster for a W-issue per cluster architecture. Two register file organizations can be used for a multithreaded VLIW architecture, namely, shared and partitioned. Figure 10 shows the shared and partitioned organization of the register files for a 2-Thread N-cluster architecture. The shared organization has a single register file with twice the registers while the partitioned organization has an individual register file for each thread. Both organizations have similar power, area and delay characteristics [4]. A detailed discussion of pros and cons of the two designs is beyond the scope of this paper.

A shared register file organization cannot be used with split-issue because the sharing of the ports limits the number of simultaneous writes. More write ports can be added to the shared register file but doing so will incur a non-trival hardware cost. On the other hand, the partitioned register file organization already provides the requisite register file ports. Hence, the paritioned register file organization is assumed for our experiments.

D. Memory Port Contention

Similar to the contention for register file write ports, a contention for memory ports also arises because of delayed updates to architectural state. If an instruction has a memory write in the split-issued part, the memory write writes into a buffer. The contents of the buffer are written to memory when the last part of the instruction is executed. It is possible

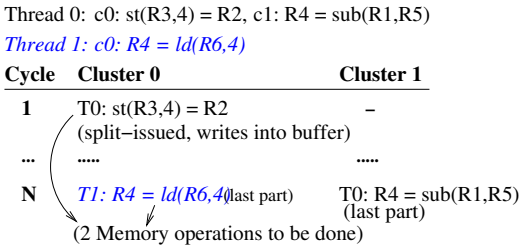


Figure 11: Memory port contention due to delayed memory writes

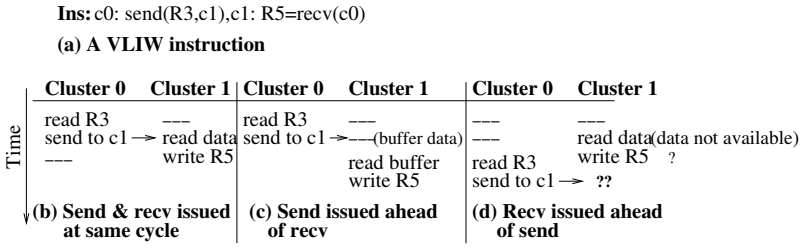


Figure 12: Issuing inter-cluster communication operations

that the last part of the instruction is issued with the last part of another instruction (belonging to a different thread). If the other instruction also has a memory operation in the same cluster as the split-issued part of the former instruction, it may happen that more memory operations have to be done than the number of memory ports available at a cluster at that cycle.

Figure 11 shows an example illustrating this issue on a 2-cluster machine with 1 memory port per cluster. The figure shows 2 instructions, each belonging to a different thread. Thread 0 has 2 operations, a memory write in cluster 0 and an alu operation in cluster 1. Thread 1 has only 1 memory read in cluster 0. At cycle 1, the memory write operation of Thread 0 is split-issued. Since the memory write is split-issued, it does not write into memory but into the buffer. The data in the buffer is committed to memory only when the last part is executed. At cycle N, the memory operation of Thread 1 and the last part of Thread 0 (i.e. alu operation) are issued. Now, two memory operations have to be performed at cluster 0 (the pending memory write of Thread 0 and the memory read of Thread 1) when the two instructions reach the execution pipeline stage. However, both memory operations cannot be performed at the same cycle, as there is only 1 memory port per cluster. To solve this issue, if such a collision for memory port is detected, the pipeline is stalled till all the memory operations have been performed.

E. Issues with Inter-cluster Communication

The experimental platform used in this paper, VEX, uses inter-cluster send and recv operations to transfer data across clusters. Send operation reads a register from its corresponding register file and sends it to another cluster over the inter-cluster communication network. Recv operation reads the data sent from the inter-cluster communication network and writes it into the corresponding register file. According to VEX semantics, send and recv operations should be simultaneously issued. However, as a consequence of split-issue, they might get issued in different cycles, which can result in an incorrect transfer of data.

Figure 12 shows an example highlighting this issue. For simplicity, the VLIW instruction shown in Figure 12(a) has only two operations, a send operation in cluster 0 and a recv operation in cluster 1. The send operation reads register R3 from register file of cluster 0 and sends it to cluster 1. The recv operation in cluster 1 writes the received data to register R5 of cluster 1. Figure 12(b) shows the normal execution of the two operations. If send is issued earlier than recv, the data arrives at cluster 1 before recv is executed. A simple solution for this issue is to buffer the data till recv is executed, as shown in 12(c). However, if recv is issued ahead of send, as shown in Figure 12(d), the data is not available when recv executes, resulting in an incorrect transfer. Hence, while send can be ahead of recv, recv cannot be issued ahead of the corresponding send.

Note that recv performs 2 functions: Read the value from interconnection network and then, write the value to the destination register. None of these functions require using a particular FU or a complex hardware. A solution to the early issue of recv can be the following: If recv detects that data is not available when reading from inter-cluster communication network, it simply saves the destination register number to a buffer. When the data arrives later, the data is written to the corresponding register. This requires usage of the partitioned register file organization to guarantee the availability of a write port to the register file.

VI. Performance Evaluation

A. Experimental Setup

Experiments have been done in a 16-issue, 4-cluster architecture configuration (i.e. 4-issue per cluster). All the experiments assume a single-level 64KB, 4-way set-associative, 20-cycles[5] miss penalty cache architecture for both ICache and DCache (No L2 cache).

We have used a set of MediaBench [18] and SpecInt 2000 [19] applications. We have also included production color space conversion [20], imaging pipeline [14] used in high performance printers, inverse discrete cosine transform (used in various codecs) [21] and H.264 encoder [22]. The benchmarks are shown in Figure 13(a). For each benchmark, columns IPC_r show the average IPC when a real memory with cache misses is used and and IPC_p show the average

[5]Assuming a target processor frequency of 400MHz and a DRAM latency of 50 ns for critical word transfer.

Benchmarks	*ILP Degree*	*Description*	IPC_r	IPC_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
x264	h	H.264 encoder	3.89	4.04
idct	h	Inverse DCT	4.79	5.27
colorspace	h	Colorspace Conversion	5.47	8.88

(a) **Benchmarks**

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	x264
llhh	mcf	blowfish	x264	idct
lmhh	gsmencode	g721encode	imgpipe	colorspace
mmhh	djpeg	g721decode	idct	colorspace
hhhh	x264	idct	imgpipe	colorspace

(b) **Workloads**

Figure 13: Benchmarks and Workloads Used

IPC obtained when a perfect memory with no cache misses is used. Benchmarks are classified by their IPC_p in three categories: high IPC (colorspace, imgpipe, idct and x264), medium IPC (g721encode, g721decode, cjpeg and djpeg) and low IPC (mcf, bzip2, blowfish and gsmencode). This classification is shown in column $ILP\ Degree$ as `l` (low IPC), `m` (medium IPC) and `h` (high IPC).

The workloads used are listed in Figure 13(b). In order to select appropriate thread configurations, we have combined benchmarks with different IPC degrees, attempting to cover representative combinations. For instance, playing a dvd requires multiple threads for decryption (low ILP), video decoding (high ILP), audio decoding (medium ILP) etc. along with the operating system threads (low ILP). Column labeled as $ILP\ Comb$ indicates these IPC combinations. For example, configuration `llhh` in Figure 13(b) has two benchmarks with low IPC and two benchmarks with high IPC, configuration `llmm` has two benchmarks with low IPC, and two benchmarks with medium IPC and configuration `lmhh` has one benchmark with low IPC, one benchmark with medium IPC and two benchmarks with high IPC.

We carried out the experiments by arranging the workloads in a multitasking environment. The number of threads supported by the processor is exposed as virtual CPUs and the task scheduler schedules as many threads to run as the number of virtual CPUs, with a timeslice of 5 million cycles. After the expiry of the timeslice, a context switch takes place and the running threads are replaced by other threads from the workload. 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. To alleviate bias and to improve fairness, replacement threads are picked at random from the workload after the context switch. The execution packet is formed by merging instructions from as many threads as possible according to their priority. First, the instruction from the highest priority thread is selected; then, the instruction from the next highest priority thread is tried to be merged in the execution packet, and so on. A different priority is assigned to each selected thread in a round robin way every cycle.

The workloads are executed till one thread completes executing 200 million VLIW instructions (1 VLIW instruction = 1 to 16 RISC instructions). If any of the benchmarks finishes before some thread can finish executing 200M VLIW instructions, then that benchmark is respawned again. All benchmarks except mcf and bzip2 are relatively short (30-100M VLIW instructions) and run to completion atleast once. Thanks to the respawning of benchmarks, the performance results are very stable and do not require use of an IPC stabilization technique like FAME [23].

B. Results

In this section, we present the experimental results obtained by implementing split-issue both at cluster-level and operation-level. Two different architectural configurations have been evaluated in this paper to analyze the impact of inter-cluster communication operations on split-issue.

No split communication: In this configuration, instructions with inter-cluster communication operations are not split at all. No spliting of instructions with inter-cluster communication operations insures that compiler assumptions are never violated. Thus, no measures are required to maintain correctness.

Always split: This configuration allows splitting of instructions with inter-cluster communication operations. Splitting these instructions can violate the compiler assumptions because recv and send operations may execute at different cycles. In particular, the concern is with recv executing ahead of send. As a result, this configuration requires extra hardware to avoid breaking execution semantics as discussed previously in Section V-E.

First, we discuss the performance results obtained when instructions can be merged only at cluster-level. Only cluster-level split-issue is applicable when instructions are merged at cluster-level. Figure 14 shows the speedups (measured in terms of IPC) obtained by CCSI (cluster-level merging with cluster-level split-issue) over a 2-Thread and a 4-Thread CSMT (cluster-level merging but no split-issue) machine respectively. In the figure, label 'AS' denotes the 'Always split' architectural configuration and label

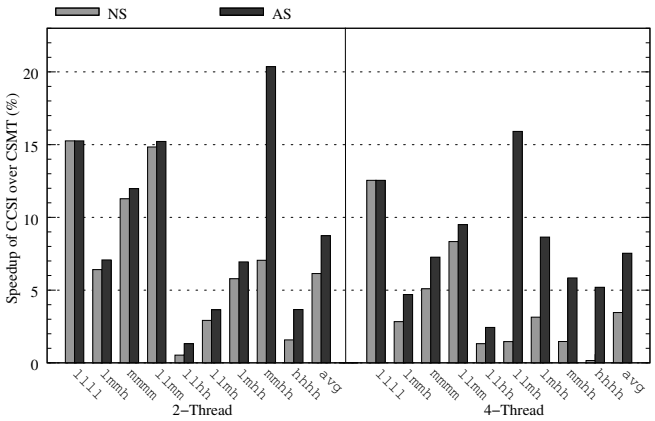


Figure 14: Cluster-level split-issue (CCSI) speedups over CSMT

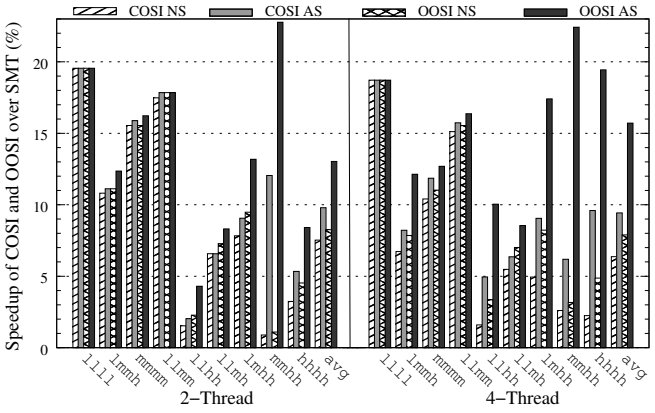


Figure 15: Speedups over SMT with cluster-level (COSI) and operation-level split-issue (OOSI)

'NS' represents the 'No split communication' architectural configuration. When splitting of instructions with inter-cluster communication operations is not permitted (NS), CCSI achieves, on an average, a speedup of 6.1% over a 2-Thread CSMT architecture and 3.5% over a 4-Thread CSMT architecture. For particular cases like llll, speedups as high as 15.1% are obtained. When splitting of inter-cluster communication operations is allowed (AS), an average speedup of 8.7% over a 2-Thread and 7.5% over a 4-Thread CSMT machine is achieved on an average. For particular cases like mmhh, a speedup of 20.3% is obtained over a 2-Thread CSMT machine.

Next, we discuss the performance results obtained when instructions are merged at operation-level. In this case, both operation-level split-issue (OOSI) and cluster-level split-issue (COSI) techniques are applicable. Figure 15 shows the speedups obtained by OOSI and COSI over a 2-Thread and a 4-Thread SMT (operation-level merging but no split-issue) machine. In the figure, label 'AS' denotes the 'Always split' architectural configuration and label 'NS' represents the 'No split communication' architectural configuration. When splitting of instructions with inter-cluster communication operations is not permitted (NS), COSI achieves ,on an average, a speedup of 7.5% and 6.4% over a 2-Thread and a 4-Thread SMT machine respectively. Using OOSI achieves higher performance improvements, on an average 8.2% over a 2-Thread and 7.9% over a 4-Thread SMT machine.

When splitting of instructions with inter-cluster communication operations is permitted (AS), COSI speedups increase to 9.8% over a 2-Thread SMT machine and 9.4% over a 4-Thread SMT machine. Speedup as high as 19.5% is achieved for particular cases like llll for 2-Thread SMT machine and 18.7% for a 4-Thread SMT machine. For OOSI, performance improvements increase to 13% and 15.7% on an average. For particular cases like mmhh, OOSI achieves speedups as high as 22.7% over a 2-Thread SMT machine and 22.4% over a 4-Thread SMT machine.

Note that when splitting of instructions with inter-cluster communication operations is not allowed (NS), performance improvement obtained for workloads containing benchmarks with high IPC is much lower than the one obtained when splitting of instructions with inter-cluster communication operations is allowed (AS). This holds true for any split-issue scheme used (CCSI, COSI and OOSI). For instance, for workload `mmhh`, using CCSI results into a performance gain of 7.4% on a 2-Thread CSMT machine for 'No split communication' model. For the 'Always split' model, the speedup increases almost threefolds to 20.3%. The large difference arises because high IPC benchmarks use inter-cluster communication operations more frequently than the low and medium IPC benchmarks. Constrain on splitting instructions with inter-cluster communication operations leads to an infrequent use of split-issue making it harder to fill the empty issue-slots. Hence, a significant difference in performance is observed.

Finally, we present an absolute performance comparison of all the multithreading techniques for all architectural configurations evaluated in this paper. For ease of comparison, Figure 16 shows the average performance of all the techniques for a 2-Thread and a 4-Thread machine. The first thing to note is that by use of split-issue, cluster-level merging (CCSI AS) has practically the same performance (in fact, slightly better) as operation-level merging (SMT) for 2-Thread machine. Even on a 4-Thread processor, the performance difference between cluster-level merging (CSMT) and operation-level merging (SMT) decreases from 27% to only 13% when split-issue is used (CCSI AS). Since cluster-level merging is much cheaper to implement than operation-level, this makes cluster-level merging even more attractive.

Next we focus on the performance difference between operation-level (OOSI) and cluster-level split-issue (COSI) when instructions are merged at operation-level. In general, COSI performance is always lower than OOSI. However, the performance difference is small. When splitting instructions

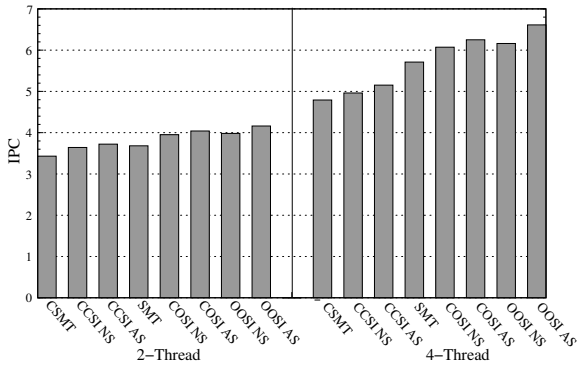


Figure 16: Performance of all multithreading techniques

with intercluster communication operations is not permitted, operation-level split-issue (OOSI NS) has an average performance advantage of only 0.7% over cluster-level split-issue (COSI NS) for 2-Thread, and 1.4% for a 4-Thread configuration. When splitting of intercluster communication operations is permitted, the performance advantage of operation-level split-issue (OOSI AS) over cluster-level split-issue (COSI AS) is only a little higher, on an average 2.7% for a 2-Thread and 5.7% for a 4-Thread configuration.

In conclusion, using cluster-level split-issue improves the multithreading performance of clustered VLIW processors at a low hardware overhead. Cluster-level split-issue achieves similar performance as previously prosposed split-issue technique, operation-level split-issue, but at a much lower complexity. Implementing operation-level split-issue requires dynamic scheduling and hence, it requires complex structures. On the other hand, the changes required by cluster-level split-issue are much smaller in nature and does not increase the complexity of the processor significantly. Hence, cluster-level split-issue is a more cost effective solution than operation-level split-issue for clustered VLIW processors.

VII. Conclusions

Clustered VLIW processors are an attractive choice in embedded domain due to their low power and low cost advantages. Simultaneous MultiThreading (SMT) reduces horizontal and vertical waste in the processor by simultaneously issuing instructions from multiple threads. The restriction to issue a VLIW instruction in its entirety restricts the opportunities to reduce horizontal waste.

Operation-level split-issue, removes the constrain of having to issue an instruction in entirety and allows a flexible issue of operations of a VLIW instruction. Operation-level split-issue does a dynamic scheduling of operations of VLIW instruction and increases SMT performance. However, implementing dynamic scheduling requires complex hardware which is not practical for area and power sensitive embedded clustered VLIW processors. Cluster-level split-issue, the technique proposed in this paper, splits an instruction only at a cluster-level boundary. This eliminates the need for dynamic scheduling and has a much cheaper hardware implementation. Besides, the hardware changes required to implement cluster-level split-issue are quite small in nature and can be easily incorporated.

Experimental results show that cluster-level split-issue significantly improves performance. When instructions are merged at cluster-level, a performance improvement 8.7% is obtained over a 2-thread and 7.5% over a 4-thread processor (operation-level split-issue is not applicable for cluster-level merging). With instruction merging at operation-level (traditional SMT), a performance improvement of 9.8% is obtained over a 2-thread and 9.4% over a 4-thread SMT processor. In particular cases, performance improvements as high as 18.7% are achieved. Further, cluster-level split-issue, despite being more restrictive than operation-level split-issue, achieves similar performance. On an average, cluster-level split-issue performance is within 2.7% for a 2-thread processor and 5.7% for a 4-thread processor when compared to the performance obtained by using operation-level split-issue. Hence, cluster-level split-issue is more cost effective and practical for SMT clustered VLIW processors than operation-level split-issue.

VIII. Acknowledgements

This work is supported by Spanish Ministry of Science and Technology under contract CICYT TIN2007-60625, FI grant from AGAUR/Generalitat de Catalunya, SARC (Scalable computer ARChitecture) Project and HiPEAC European Network of Excellence.

References

[1] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood, "Lx: A Technology Platform for Customizable VLIW Embedded Processing," in *ISCA*, 2000, pp. 203–213.

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

[3] N. Seshan, "High VelociTI Processing," *IEEE Signal Processing Magazine*, vol. 15, no. 2, pp. 86–101, March 1998.

[4] S. Rixner, W. J. Dally, B. Khailany, P. R. Mattson, U. J. Kapasi, and J. D. Owens, "Register Organization for Media Processing." in *HPCA*, 2000.

[5] A. Mikschl and W. Damm, "MSparc: A Multithreaded Sparc," in *Euro-Par, Vol. II*, 1996, pp. 461–469.

[6] W.-D. Weber and A. Gupta, "Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: preliminary results," in *ISCA*, 1989.

[7] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. J. Smith, "The Tera computer system," in *ICS*, 1990.

[8] B. J. Smith, " Architecture and Applications of the HEP Multiprocessor Computer System," in *SPIE*, 1981, pp. 241–248.

[9] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," in *ISCA*, 1995.

[10] M. Gupta, F. Snchez, and J. Llosa, "Cluster-Level Simultaneous MultiThreading for VLIW Processors," in *ICCD*, 2007.

[11] B. R. Rau, "Dynamically Scheduled VLIW Processors," in *MICRO*, 1993, pp. 80–92.

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

[13] S. Palacharla, N. Jouppi, and J. Smith, "Complexity-effective superscalar processors," in *ISCA*, 1997, pp. 206–218.

[14] VEX Toolchain, "http://www.hpl.hp.com/downloads/vex/."

[15] 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*, vol. 7, no. 1-2, pp. 51–142, 1993.

[16] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. Computers*, vol. 30, no. 7, pp. 478–490, 1981.

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

[18] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," in *MICRO*, 1997, pp. 330–335.

[19] J. L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *IEEE Computer*, vol. 33, no. 7, pp. 28–35, 2000.

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

[21] Inverse discrete cosine transform, taken from ffmpeg, "http://ffmpeg.mplayerhq.hu. last consult april 2008."

[22] x264 - a free h264/avc encoder, "http://www.videolan.org/developers/x264.html. Last consult April 2008."

[23] J. Vera, F. Cazorla, A. Pajuelo, O. Santana, E. Fernandez, and M. Valero, "FAME: FAirly MEasuring Multithreaded Architectures," in *PACT*, 2007.