Modulo Scheduling for a Fully-Distributed Clustered VLIW Architecture

Jesús Sánchez and Antonio González

Dept. of Computer Architecture
Universitat Politècnica de Catalunya
Barcelona - SPAIN

E-mail: {fran,antonio}@ac.upc.es

Abstract

Clustering is an approach that many microprocessors are adopting in recent times in order to mitigate the increasing penalties of wire delays. In this work we propose a novel clustered VLIW architecture which has all its resources partitioned among clusters, including the cache memory. A modulo scheduling scheme for this architecture is also proposed. This algorithm takes into account both register and memory inter-cluster communications so that the final schedule results in a cluster assignment that favors cluster locality in cache references and register accesses. It has been evaluated for both 2- and 4-cluster configurations and for differing number and latencies of inter-cluster buses. The proposed algorithm produces schedules with very low communication requirements and outperforms previous cluster-oriented schedulers.

1. Introduction

Technology projections point to wire delays as being one of the main hurdles for improving instruction throughput of future microprocessors [23]. As wire delays grow relative to gate delays and feature sizes shrink, the percentage of on-chip transistors that can be reached in a single cycle will decrease, and microprocessors will become *communication bound* rather than *capacity bound* [1][14].

Techniques to solve this problem at all levels, from applications to technology, will be crucial for performance. Clustering is an effective microarchitectural approach to mitigate the negative effect of wire delays. The main idea is to have a hierarchical organization of the interconnection wires such that units that communicate frequently are interconnected through short and fast wires. On the other hand, units that rarely communicate can use longer and slower wires. In other words, the microarchitecture exploits what we may call *communication locality*. Several commercial microprocessors have adopted this approach, such as the Alpha 21264 [10], which is a superscalar processor, but this trend is even more common for VLIW processors used in the embedded/DSP domain. Examples of the latter are Texas Instrument's TMS320C6000 [24], Equator's MAP1000 [15] and Analog's TigerSharc [8].

Clustering can be applied to different parts of the microarchitecture. Cluster microarchitectures proposed so far, both in the commercial and research arena, distribute the functional units and register files, but the data cache is considered a centralized resource. This centralized organization challenges the scalability of these architectures. Besides, some studies point out that the access time (in number of cycles) to the memory structures is likely to increase with future technologies, even when their capacity is kept constant [1]. This suggests that short latency memory structures should be even smaller than they are today. Because of these two reasons, we believe that a distributed cache memory architecture is key for increasing the performance of future microarchitectures.

In this work we propose a clustered VLIW microarchitecture with a distributed cache memory. This architecture has all the resources distributed: instruction fetch, execute and memory units. It resembles very much a multiprocessor, with the exception that all the clusters progress in a lockstep mode, and inter-cluster register communications are controlled by the compiler by means of certain fields in the ISA. Because of this resemblance we refer to this architecture as a *multiVLIWprocessor*.

The effectiveness of this microarchitecture strongly depends on the ability of the compiler to generate code that balances the workload of the different clusters and result in few inter-cluster communications. In this work we propose a modulo scheduling technique for *multiVLIWprocessors*. The proposed scheduler includes some heuristics for minimizing inter-cluster register communication, based on the information provided by the data dependence graph. Besides, it implements a powerful memory locality analysis based on *Cache Miss Equations* [9], which guides the scheduling of memory instructions with the objective of minimizing inter-cluster memory communications.

0-7695-0924-X/00 $10.00 © 2000 IEEE

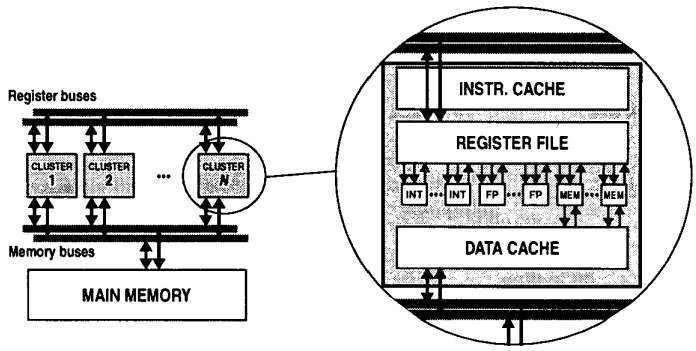


Figure 1. Microarchitectures of a MultiVLIWProcessor

Some previous work related to scheduling of instructions for clustered VLIW architectures can be found in the literature for non-cyclic [6][4][11][18] and cyclic code [17][7][22], but to the best of our knowledge this is the first study that deals with a clustered VLIW architecture that has a distributed data cache.

The rest of this paper is organized as follows. Section 2 describes the architecture of the *multiVLIWprocessor* and some basic background on modulo scheduling. An example that motivates the proposed algorithm is shown in Section 3. In Section 4, the proposed algorithm is described and Section 5 shows performance results obtained for different configurations. Finally, the main conclusions of the work are drawn in Section 6.

2. MultiVLIWProcessors

In this section we first describe the microarchitecture of *multiVLIWprocessors* and then we review some basic concepts of modulo scheduling for the proposed architecture.

2.1. Microarchitecture

Our base architecture (see Figure 1) is composed of several clusters, each one executing a fixed part of each VLIW instruction. All clusters work in lockstep mode, i.e., any stall in one cluster also stalls the other clusters. Every cycle, all clusters fetch their corresponding parts of a new VLIW instruction from their local instruction caches. Each cluster consists of several functional units, a register file and a local data cache memory in addition to the local instruction cache. Functional units can be of three different types: integer arithmetic, floating-point arithmetic or memory access. For the sake of simplicity, we consider that all clusters are homogeneous (i.e., with the same number and type of functional units), but the proposed techniques can be generalized for heterogeneous clusters.

Register values generated by one cluster and needed by another one are communicated through a set of buses that are shared by all clusters (called *register buses*). A value that is put in a register bus can come from either the local register file or the output of a functional unit through a short-circuit. On the other hand, a value that is read from the bus can be stored in a register file, feed a functional unit or both. Thus, instruction register operands can be read from either the local register file or any bus, and instruction results can be written into the register file and to any register bus. All register communication operations are explicitly encoded in the appropriate fields of the VLIW instruction, which are set at compile time. Thus, no additional hardware is needed to manage and arbitrate register buses. The detailed VLIW instruction format is shown in Figure 2. Each instruction for a particular cluster consists of the following fields. An operation for each functional unit in that particular cluster (FUj) and the source (IN BUS) and target (OUT BUS) of the bus (there are as many IN/OUT fields as number of buses). The IN BUS field indicates, if necessary, the register in the local register file in which the value that is in IRV has to be stored. The IRV (*Incoming Register Value*) is a special register in each cluster that latches the value that comes from the bus. The OUT BUS field indicates from which local register a value has to be

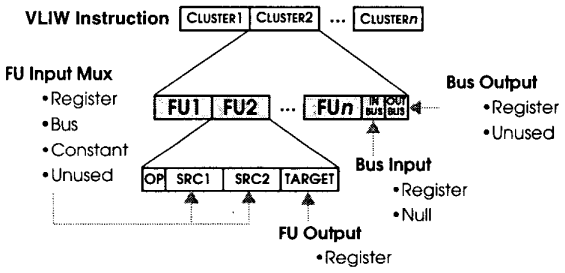


Figure 2. VLIW instruction format

issued to the bus, if any. If the register is being written in that cycle, the data will be bypassed from the output of the corresponding functional unit. Since a bus is a resource shared by all the clusters, when one particular cluster places a data on the bus (OUT BUS), this bus will be busy during the entire bus latency and no other instruction can use it (a bus is considered by the scheduling algorithm as another resource in the reservation table).

Regarding memory accesses, a load/store issued by a cluster first tries its local L1 data cache. If the data is found, the access is satisfied with minimum latency. Otherwise, the hardware tries the cache of the other clusters or, finally, the access is solved by the main memory. Both local memories and main memory are interconnected through one or several buses (that are called *memory buses*). As the cache is physically partitioned among the clusters, coherence among the local caches and the main memory has to be kept. For this reason, a snoopy MSI protocol [5] has been implemented. This protocol is completely transparent to the ISA, and further, both the coherence and the bus arbitration are managed by the hardware. When a memory access misses in its local cache, the miss request is queued in a local MSHR (*Miss information/Status Handling Register*) structure, since the L1 data cache is non-blocking [12]. Then, the access has to compete for a free memory bus in order to access a remote cache or the main memory.

All the dependences with memory operations are dynamically checked, since the scheduler may have considered an optimistic latency for these instructions (i.e., hit in the local cache). If any dependence is not met, the dependent instruction stalls in all clusters until the hazard is resolved.

2.2. Background on Modulo Scheduling

Software pipelining is a very effective technique to statically schedule loops. The most popular scheme to perform software pipelining is called modulo scheduling [20][13]. The two main parameters that statically characterize a modulo scheduled loop are the *initiation interval* (II) and the *stage count* (SC). The former reflects the number of cycles that a kernel iteration takes (assuming no stalls), whereas the latter shows how many iterations are overlapped, and determines the length of the prolog and epilog.

For a clustered VLIW architecture, both II and SC can be affected by inter-cluster register communications. If the communication buses become saturated, a higher II is required. Moreover, communication operations may increase the length of the schedule, and therefore the SC may be increased. Thus, the IPC of a clustered VLIW architecture will be lower than that of an equivalent unified VLIW architecture with the same resources in general. On the other hand, a clustered architecture may reduce the critical delays such as the register file access time and the bypass latency [19], and allow for faster clock rates.

For this paper, which focuses on modulo scheduling for *multiVLIWprocessors*, the number of cycles needed to execute a particular modulo scheduled loop can be modeled through the following expression [21]:

$$NCYCLE_{Total} = NCYCLE_{Compute} + NCYCLE_{Stall}$$

Where $NCYCLE_{Compute}$ represents a fixed number of cycles that depends on the particular static scheduling produced by the compiler. During these cycles the processor is doing useful (or at least scheduled) work. $NCYCLE_{Stall}$ represents the number of cycles where the processor is stalled and depends on several factors as we detail below. The value of $NCYCLE_{Compute}$ can be computed before executing the loop if the number of times the loop is executed (NTIMES) and the number of iterations of each execution (NITER) are known, as shown by the next expression:

$$NCYCLE_{Compute} = NTIMES * ((NITER + SC - 1) * II)$$

The value of $NCYCLE_{Stall}$ cannot be computed statically. It represents the number of stall cycles due to incomplete information managed by the compiler. For instance, some memory instruction latencies may be unknown since the compiler does not know whether they will hit in the first level cache. If the value loaded by a memory instruction feeds another operation (i.e., the latter depends on the former) but the latter was scheduled using an underestimation of the memory latency, it will stall until the memory access is finished. In the assumed microarchitecture, the final latency of a memory instruction depends on three factors:

- Latency of memory accesses, which depends on the memory level that satisfies the access: local cache, remote cache or main memory.
- Number of entries in the MSHR of the lockup-free caches. If there is no available entry for a new miss request, the instruction stalls until there is a free entry.
- Cycles waiting for a free bus and bus latency.

Thus, considering all of these factors, the total latency of a memory access can be represented by this formula:

$$LAT_{MemAccess} = LAT_{Cache} + MISS_{LC} * (NC_{WaitingEntry} + NC_{WaitingBus} + LAT_{MemoryBus} + \max(LAT_{Cache}, MISS_{RC} * LAT_{MainMemory}))$$

Where both $MISS_{LC}$ and $MISS_{RC}$ represent binary values that are 1 if the access misses in local cache and all remote caches respectively, or 0 otherwise. $NC_{WaitingEntry}$ represents the number of cycles that a miss access is wait-

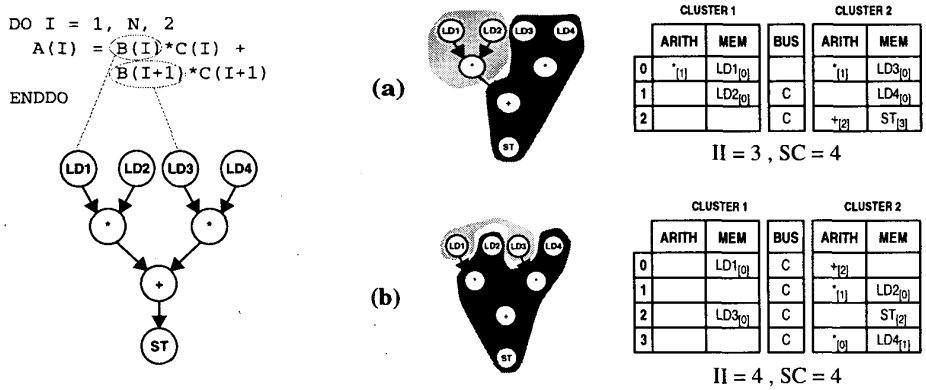


Figure 3. Motivating example

ing for an available entry in the MSHR. $NC_{WaitingBus}$ is the number of cycles that the access is waiting for a free bus. Note that a bus can be also busy for coherence operations and this is taken into account by our simulator. Finally, although we have considered $LAT_{MainMemory}$ as a fixed parameter, in the above expression note that for some references this number could be smaller if an earlier miss has already started loading the relevant cache line. This fact has also been accounted for our simulator.

3. Motivating Example for the Proposed Scheduler

The objective of this study is twofold: first, demonstrate that when the data cache is partitioned among the different clusters, the selection of the cluster where each memory instruction is scheduled is very important and can dramatically affect the final performance of a program (the same holds for register values, but this has already been shown by previous papers). Second, we propose a modulo scheduler that takes into account both register and memory intercluster communications.

In this section, we illustrate through an example how the cluster selection can affect the total number of cycles in which a code section is executed. Consider that we want to perform modulo scheduling of a loop whose code and dependence graph are shown in Figure 3. Assume the processor consists of 2 clusters, each one with its local register file and data cache (direct-mapped), and 2 functional units: one for arithmetic operations (with 2-cycle latency) and one for memory operations. There is one inter-register bus with a 2-cycle latency. The latencies for memory accesses are: 2 cycles for a local cache, 2 cycles for a bus transaction and 10 cycles for an access to main memory.

For this loop, the *minimum initiation interval* (mII) for an equivalent unified architecture with the same resources is 3 cycles. The partition and scheduling that minimizes the number of register communications between clusters and, thus, that achieves the same II as the equivalent unified architecture is shown in Figure 3(a). In this figure, the left part represents the partition of the operations between the clusters whereas the right part shows the modulo reservation table obtained after modulo scheduling. Each operation is scheduled in a particular slot and the number in brackets represents the stage at which this operation is scheduled. The usage of the register bus is also shown in this table. Whenever a bus transaction takes place, the corresponding bus time slot is reserved and it is indicated by a *C* in the reservation table.

Then, the $NCYCLE_{Compute}$ of the resulting loop can be computed as:

$$NCYCLE_{Compute(a)} = NTIMES \cdot ((N + 4 - 1) \cdot 3) = NTIMES \cdot (N + 3) \cdot 3$$

However, suppose that both arrays B and C are located in memory at a distance that is a multiple of the local cache memory size. This means that we will have ping-pong interferences between LD1 and LD2, and between LD3 and LD4. Thus, the spatial locality exhibited by the four instructions cannot be exploited and the four accesses always miss. The result is that the instruction(s) that consume the memory values suffer many stalls. In the example, the VLIW instruction that contains the multiplications cannot continue its execution until the misses are satisfied. Assuming that we have sufficient memory buses, the number of cycles that the instruction stalls is the latency of a bus transaction plus an access to main memory, since the latency to

the local cache was taken into account by the scheduler. Then, the number of stall cycles is:

$$NCYCLE_{Stall(a)} = NTIMES * N * (2+10) = NTIMES * N * 12$$

An alternative scheduling is shown in Figure 3(b). Based on the locality properties previously observed, in this second alternative cluster assignment is selected in order to take advantage of the locality exhibited by memory instructions. For this reason, LD1 and LD3 are scheduled in the same cluster in order to profit from its group reuse, and the same applies for LD2 and LD4 which are scheduled in the other cluster. In this way, ping-pong interferences are removed and we can take advantage of the spatial reuse. However, as we can see in the example, for this case two communications between register values are needed per iteration, and then the II has to be increased from 3 to 4. Thus, $NCYCLE_{Compute}$ is computed as:

$$NCYCLE_{Compute(b)} = NTIMES * ((N + 3 - 1) * 4) = NTIMES * (N + 2) * 4$$

However, the miss rate of LD3 and LD4 is 25% (assuming eight data elements per cache block), and LD1 and LD2 always hit (excepting the first iteration). Thus, the number of stall cycles is:

$$NCYCLE_{Stall(b)} = NTIMES * N * (2*(2+10)* 0.25) = NTIMES * N * 6$$

Then, putting all together, we have that the total number of cycles in both strategies as:

$$NCYCLE_{Total(a)} = NTIMES * (15 * N + 9)$$

$$NCYCLE_{Total(b)} = NTIMES * (10 * N + 8)$$

Therefore, we can conclude that the second strategy, which takes into account both register and memory communications, achieves a schedule that is 1.5 times faster than the original one, which is optimized only for register communications.

4. Register and Memory Communication-Aware Modulo Scheduler

In this section we present a modulo scheduler that tries to minimize both register and memory inter-cluster communications and at the same time balance the workload. We first review a previously proposed scheduler, which is very effective at minimizing register communications, and which we will use as a baseline for comparisons. Then, we present the data locality analysis framework that is used by the scheduler. Finally, the modulo scheduler is described.

4.1. Baseline Algorithm

We use as the baseline algorithm the one proposed in our previous work [22], which was shown to be very effective at minimizing register communications and maximizing the workload balance. In that work, the target architecture was similar to the one proposed in Section 2.1, but in that case all clusters accessed a shared L1 cache. Below, we briefly review the algorithm proposed there. For more details, the interested reader is referred to the original paper [22].

The algorithm employs a unified assign-and-schedule approach, that is, cluster selection and scheduling of operations is done in a single step. The heuristic for selecting a cluster is the number of edges that exit from the dependence subgraph corresponding to all the nodes already scheduled in a particular cluster. This value represents a measure of the number of register communications. An attempt is made to schedule an operation (i.e., a node in the dependence graph) in all the clusters in which there is an available slot. The one chosen is the one in which the best profit from output edges is achieved (that is, the difference between output edges before and after including this operation in the partial schedule). All the operations are scheduled using the same algorithm and following a particular order that is crucial for performance. If an instruction cannot be scheduled (because no issue slot is available, or there are not enough registers, or the register buses are saturated), the II is increased and the whole process is re-started (except the ordering).

4.2. Overview of the Cache Miss Equations

Cache Miss Equations (CME) is an analytical framework to model the cache behavior that is very accurate for codes that make use of scalar variables and affine[1] array references, which is very common in numeric applications. This framework was proposed by Gosh, Martonosi and Malik [9]. CME describes the precise relationship among the iteration space, array sizes, base addresses and cache parameters for a loop nest.

A direct solution of the CME is an NP problem, which makes it infeasible for many practical cases. The problem can basically be stated as counting integer points inside an exponential number of polyhedra. However, Bermudo *et al.* [3] proposed some techniques to speed-up the counting process by exploiting some intrinsic properties of the particular type of polyhedra generated by the CME. Further, Vera *et al.* [25] proposed a sampling scheme in order to estimate the solution by means of confidence intervals.

1. An array reference is affine if the expressions that indicate the referenced element in each dimension are linear functions of the loop induction variables.

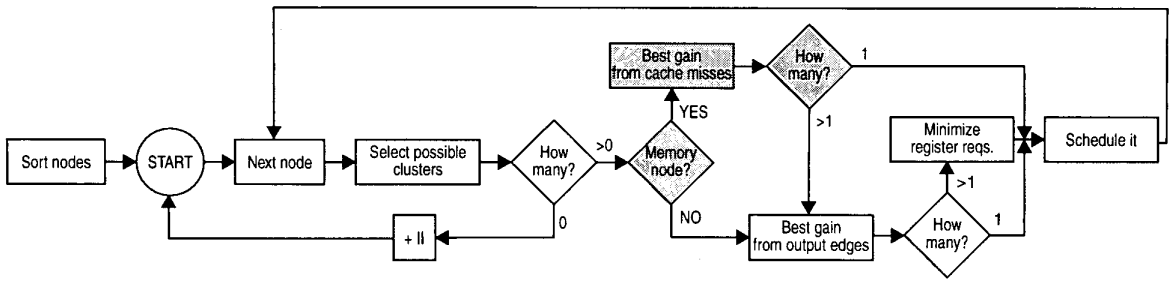


Figure 4. *RCMA* modulo scheduling step by step

These two techniques together drastically reduce the computing time to just about a few seconds per loop for most programs, and then the time required to compute and solve the equations is comparable to the time required by other typical optimizations of the compiler. In this paper, we use this implementation of the CME to estimate the amount of reuse that is exploited by any subset of memory instructions. CME will allow the scheduler to estimate the amount of memory communications among clusters or between clusters and main memory. The scheduler uses this information to guide its scheduling decisions. For instance, given a memory instruction, it is beneficial to schedule it in a cluster where there already are other instructions from which it reuses data (group reuse). On the other hand, it is detrimental to schedule the instruction in a cluster where there already are other instructions that cause many cache conflicts with the current one. CME allow the schedule to quantify the amount of reuse and conflicts among any group of instructions of the same loop nest. CME are used to produce the following statistics:

- The number of misses incurred by a set of memory references for a particular cache configuration (capacity, block size and associativity)
- The miss ratio of a particular memory instruction in this set.

4.3. Scheduler for a Distributed Cache

The proposed algorithm is called *RMCA* (which stands for *Register and Memory Communication-Aware*) modulo scheduling. It is an evolution of the algorithm reviewed in Section 4.1 and its main steps are depicted in Figure 4 (new features are shown in gray boxes). All nodes in the data dependence graph are first sorted according to the criteria used by the original paper [22]. This ordering minimizes the number of nodes that have both predecessors and successors in the set of nodes that precede it in the order. Then, cluster selection and scheduling is performed in a single step following that order. However, there is now a distinction between two types of nodes: (a) memory operations, and (b) non-memory operations. For operations of the latter group, the algorithm does not change. However, when a memory operation is scheduled, a different strategy is used. Instead of choosing the cluster where the gain from output register edges is maximized, the cluster selection depends on the profit from cache misses. In other words, each time a memory operation is scheduled, all clusters are tried, and for each one, the number of cache misses contributed by memory operations scheduled in that cluster, before and after introducing the current operation, is computed through the CME. Then, the cluster(s) where this gain is maximized is chosen. If more than one cluster is optimal with respect to cache misses, the scheduler selects one of them using the same strategy as for non-memory operations. Although the solver of the CME have to be repeatedly invoked, the method is very fast due to the optimizations mentioned in Section 4.2., and the time required by the scheduler is a small percentage of the total compilation time.

This algorithm tries to minimize the number of cache misses, and thus it attempts to minimize the inter-cluster memory communications. However, the latency of these communications can be hidden by scheduling some load instructions using the cache-miss latency (binding prefetching, as proposed in [21]). When a load is scheduled using the cache-miss latency, the operation that consumes the data read by the load will not be stalled because it is scheduled assuming the worst-case latency. However, scheduling instructions using a larger latency can have a negative effect on both register pressure and length of the schedule. On one hand, the lifetime of the load destination register is increased. On the other hand, the II can be increased if this instruction belongs to a recurrence and this increased latency makes the recurrence the most restrictive constraint on the II. Besides, the length of the schedule for a single iteration may increase, which may cause an increase in the SC, which in turn affects the durations of the prologue and epilogue. Therefore, as shown in [21], it may

be much more effective to schedule with a miss latency only those loads that are likely to miss. This can be done as long as the latency does not increase the II with respect to the schedule produced when loads are scheduled with a hit latency. Thus, the proposed scheme includes another step: once the target cluster of an instruction is determined, it is scheduled using the cache-miss latency if the miss ratio of this instruction in this particular cluster (considering the partial schedule produced so far) is greater than a certain threshold, and provided that this latency does not increase the II if the operation is in a recurrence. The assumed miss latency is the time to access main memory, that is, $LAT_{Cache} + LAT_{MemoryBus} + LAT_{MainMemory}$ (note that we do not consider the memory bus contention since it is not known at this moment, although it could be estimated).

Note that with this scheme some memory instructions are scheduled with the miss latency even if their miss ratio is lower than 100%. This may happen for instance for instructions with spatial locality. In this case, loop unrolling could be used to generate multiple instances of the same instruction such that one of them always miss and the other always hit [16]. However, we have not considered this optimization in this paper.

5. Performance Results

This section analyzes the performance of the proposed scheduler. The main performance metric that we use is the number of cycles executing instructions of modulo scheduled loops. Note that this metric does not include the effect of clustering on the cycle time, thus, differences observed for different schedulers and the same architecture directly translate into differences in execution time. However, the number of cycles for different architectures should be divided by cycle time to measure differences in execution time. Since we are concerned with differences among alternative schedulers, we prefer not to include the effect of cycle time in our metric, to isolate the effect of the schedulers. A study of the impact of clustering on cycle time can be found elsewhere [19] as well as on energy consumption [26], which is another important factor that can be reduced through clustering.

5.1. Configurations and Benchmarks

The scheduling algorithm has been evaluated for three different configurations of the *multiVLIWprocessor* architecture. These configurations are shown in Table 1.The first configuration is called *Unified* and it is composed of a single cluster with four functional units of each type (integer, floating point and memory) and a unique register file of 64 general-purpose registers.

Resources	Unified	2-cluster	4-cluster
INT / cluster	4	2	1
FP / cluster	4	2	1
MEM / cluster	4	2	1
REGS / cluster	64	32	16

Latencies	INT	FP
MEM	2	2
ARITH	1	3
MUL/ABS	2	6
DIV/SQR/TRG	6	18

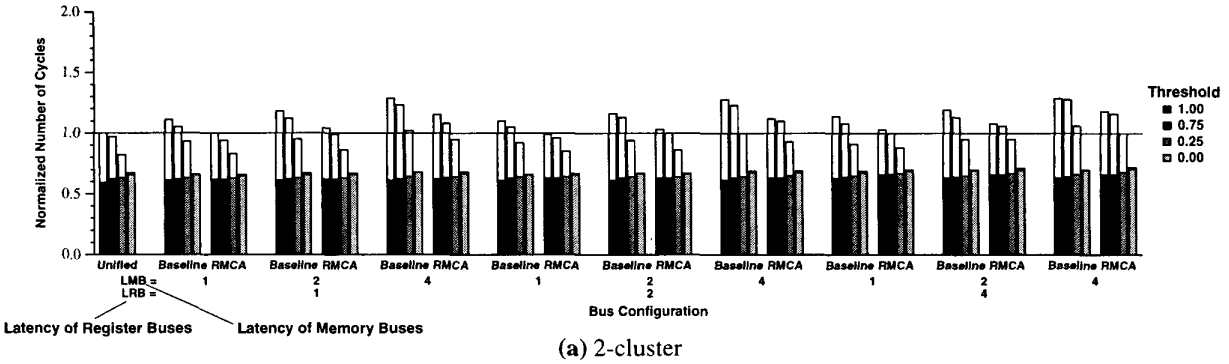
Table 1. MultiVLIWProcessor configurations and operation latencies

This configuration represents our baseline. Both the *2-cluster* and *4-cluster* configurations have the register file partitioned (into two and four partitions respectively). The former has 2 functional units of each type and 32 register per cluster and the latter includes 1 functional unit of each type and a register file of 16 registers per cluster. The three configurations are 12-way issue.

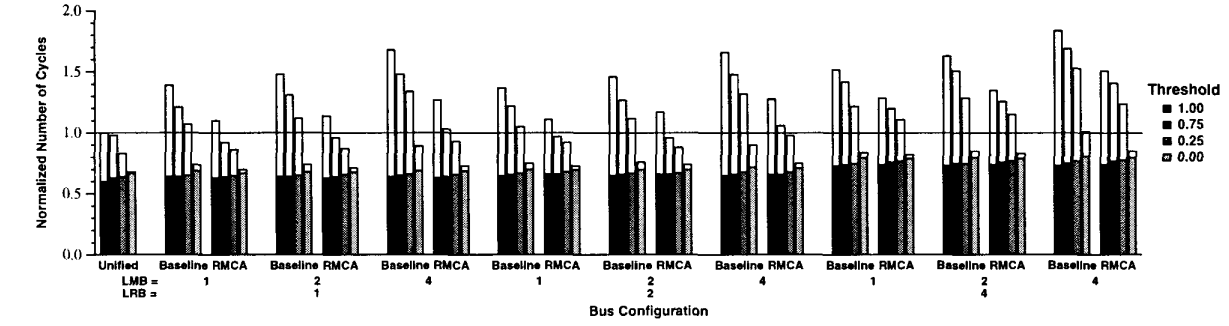
For all configurations, the total L1 cache size is 8KB, divided into equal-sizes among the different clusters. This cache capacity is realistic for embedded/DSP processors. For instance, the TI TMS320C6711 has an L1 data cache of 4Kbytes [24]. In our architecture, each local cache is direct-mapped, non-blocking with 10 entries in the MSHR. An access to a local cache is satisfied in 2 cycles, whereas an access to main memory takes 10 cycles. For the clustered configurations we will present results for different number and latency of both register and memory buses.

The modulo scheduling algorithm has been implemented in the ICTINEO compiler [2] and some SPECfp95 benchmarks have been evaluated: *tomcatv*, *swim*, *su2cor*, *hydro2d*, *mgrid*, *applu*, *turb3d* and *apsi*. Note that modulo scheduling is an effective technique for both numeric and multimedia applications, but it is not so effective for applications such as SPECint95 due to the small number of iterations for each loop execution and the abundance of conditionals.

The performance figures shown in this section refer to the modulo scheduling of innermost loops with a number of iterations greater than four. Our measurement shows that code inside such innermost loops represents about 90% of all the executed instructions, so that the statistics for innermost loops are quite representative of the whole program. Only instructions that belong to modulo scheduled loops are taken into account by the simulator. Thus, the programs were run until the first 100 million memory instructions in these loops using the ref input data set.



(a) 2-cluster



(b) 4-cluster

Figure 5. Results obtained for an unbounded number of buses (averaged for all benchmarks)

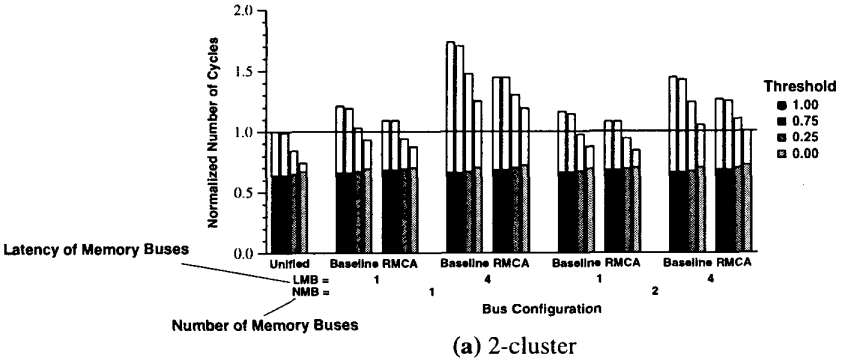
5.2. An Unbounded Number of Buses

Before considering realistic configurations, we have evaluated an architecture with an unbounded number of buses to test the performance of the proposed algorithm under extreme situations where bus bandwidth in not a problem. The remaining parameters of the architecture are those listed in Section 5.1 and the latency of the buses is parametrized. Figure 5 shows the normalized number of cycles averaged for all benchmarks, for 2 and 4 clusters and the different latencies considered. The first set of four bars represents the results for the unified configuration. The rest represent the results for the clustered configuration for different latencies of register buses (LRB - *Latency of Register Buses*) and memory buses (LMB - *Latency of Memory Buses*). For the different sets, we have evaluated two different schedulers:

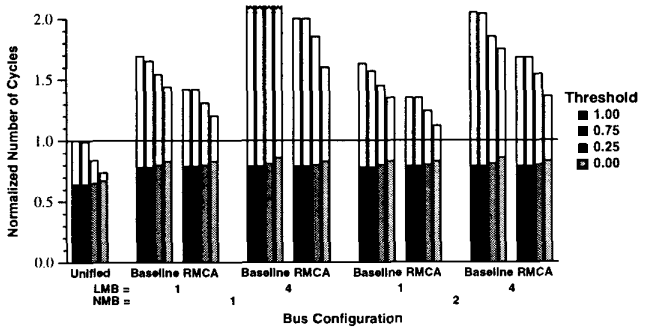
- The baseline scheduler outlined in Section 4.1, which is very effective at minimizing register communications.
- The proposed algorithm, that takes into account both register and memory communications, which is labeled as *RMCA*.

Each set of four bars represents the results obtained for different values of the cache miss threshold (from 1.00 to 0.00) that determines whether a load is attempted to be scheduled with a miss latency. Note that threshold 1.00 represents the traditional scheme, that is, using always the cache-hit latency for memory operations. On the other hand, threshold 0.00 is most similar to the one proposed in [21], where all operations that do not cause an increment in the II (due to recurrences) are scheduled using the cache-miss latency. The only difference is the locality analysis employed, which is more powerful in this paper. Each bar is split into two parts: the compute time (or $NCYCLE_{Compute}$) is the black/grey part, whereas the stall time (or $NCYCLE_{Stall}$) is the white one.

From these graphs we can see that for all configurations (number of clusters, latencies and thresholds) the scheme that takes into account memory communication (*RMCA*) outperforms the one that ignores this feature (*Baseline*). As expected, for smaller values of the threshold the compute time increases (since it may increase both the II due to register requirements, and the SC due to an increase in the length of the schedule) but the stall time decreases. Note that with a threshold of 0.00 the stall time is almost zero for all configurations and the number of cycles for the *multiVLIWprocessor* are comparable to those of the unified configuration. We can also observe that for small thresholds (0.25 or 0.00) both *Baseline* and *RMCA*



(a) 2-cluster



(b) 4-cluster

Figure 5. Results obtained when the number of buses is limited (averaged for all benchmarks)

strategies achieve similar performance, since the latency of cache misses is hidden by scheduling loads with the cache-miss latency. Nevertheless, note that for an unbounded number of buses the time waiting for a free bus ($NC_{Waiting\text{-}Bus}$) is zero, and hence, if the latency is hidden, the number of misses has no effect. However, as we will see in next section, when the number of memory buses is limited, the difference between both schemes will be notable, since the schedules produced by the *RMCA* scheme require much less communications.

5.3. Evaluation of Realistic Configurations

We have shown the potential benefits that can be achieved when memory communication are taken into account by the scheduler. In this section we study the results when a realistic inter-cluster communication network is considered.

We have evaluated configurations with a fixed number and latency of register buses (2 buses with 1-cycle latency) and for a different number and latency of memory buses. In Figure 6 we can see the results for both 2 and 4 clusters. Each set of four bars has the same meaning as in the previous section. The first set represents the results for the unified configuration. The rest are the averaged results for the different strategies (*Baseline* and *RMCA*) for 1 and 2 buses (NMB - *Number of Memory Buses*) and 1 and 4 cycles of latency (LMB - *Latency of Memory Buses*). We can observe in these graphs that, as in the unbounded study, the *RMCA* strategy outperforms the *Baseline* for all configurations. However now, for small values of the threshold, the difference between both strategies is more remarkable, mainly for 4 clusters. For the most effective threshold (0.00), the *RMCA* scheme outperforms the baseline scheduler by about 5% for 2 clusters and 20% for 4 clusters. We have observed that the reason for this difference is the time spent waiting for an available bus in order to initiate a communication. When the number of memory buses is unbounded this value is zero, because there is always an available bus. However, when the number of buses is limited, reducing the number of misses is also important since lesser the number local cache misses, lesser the number of accesses competing for a free bus time slot.

6. Conclusions

In this work we have proposed a novel microarchitecture called *multiVLIWprocessor*, which has a fully-distributed

clustered VLIW organization. The main novelty of this architecture with respect to previous proposals for clustered VLIW processors is the distributed data cache, which introduces new challenges to the instruction scheduler.

In this paper we have also presented a modulo scheduler designed for this particular architecture. This scheduler, by means of a powerful locality analysis based on the *Cache Miss Equations* and an analysis of the register data dependence graph, generates codes with very low intercluster communication requirements. We have also shown that the proposed scheduler outperforms previous schemes that just focused on register communications.

Acknowledgements

This work has been supported by the Spanish Ministry of Education under contract CICYT-TIC 511/98 and the ESPRIT Project MHAOTEU (EP24942).

References

[1] V. Agarwal, M.S. Hrishikesh, S.W. Keckler and D. Burger, "Clock Rate versus IPC: The End of the Road For Conventional Microarchitectures", in *Procs. of the 27th. Int. Symp. on Computer Architecture*, pp. 248-259, June 2000

[2] E. Ayguadé, C. Barrado, A. González, J. Labarta, D. López, S. Moreno, D. Padua, F. Reig, Q. Riera and M. Valero, "Ictineo: a Tool for Research on ILP", in *Supercomputing'96 (SC'96)*, Research Exhibit "Polaris at Work", 1996

[3] N. Bermudo, X. Vera, A. González and J. Llosa, "An Efficient Solver for Cache Miss Equations", in *Procs. of Int. Symp. on Performance Analysis and System Software*, April 2000

[4] A. Capitanio, D. Dytt and A. Nicolau, "Partitioned Register Files for VLIWs: A Preliminary Analysis of Tradeoffs", in *Procs. of 25th. Int. Symp. on Microarchitecture*, pp. 192-300, 1992

[5] D. Culler and J.P. Singh, "Parallel Computer Architecture. A Hardware/Software Approach", *Morgan Kaufmann Publishers, Inc.*, 1999

[6] J. R. Ellis, "Bulldog: A Compiler for VLIW Architectures", *MIT Press*, pp. 180-184, 1986

[7] M.M. Fernandes, J. Llosa and N. Topham, "Distributed Modulo Scheduling", in *Procs. of Int. Symp. on High-Performance Computer Architecture*, pp. 130-134, Jan. 1999

[8] J. Fridman and Zvi Greefield, "The TigerSharc DSP Architecture", *IEEE Micro*, pp. 66-76, Jan-Feb. 2000

[9] S. Ghosh, M. Martonosi and S. Malik, "Cache Miss Equations: an Analytical Representation of Cache Misses", in *Procs. of Int. Conf. on Supercomputing (ICS'97)*, pp. 317-324, July 1997

[10] L. Gwennap, "Digital 21264 Sets New Standard", *Microprocessor Report*, 10(14), Oct. 1996

[11] S. Jang, S. Carr, P. Sweany and D. Kuras, "A Code Generation Framework for VLIW Architectures with Partitioned Register Banks", in *Procs. of 3rd. Int. Conf. on Massively Parallel Computing Systems*, April 1998

[12] D. Kroft," Lockup-Free Instruction Fetch/Prefetch Cache Organization", in *Procs. 8th Int. Symp. on Computer Architecture*, pp. 81-87, 1981

[13] M. Lam, "Software pipelining: An Effective scheduling technique for VLIW Machines", in *Procs. on Conf. on Programming Languages and Implementation Design*, pp. 258-267, June 1993

[14] D. Matzke, "Will Physical Scalability Sabotage Performance Gains", *IEEE Computer, Vol. 30, No. 9*, pp. 37-39, Sept. 1997

[15] "MAP1000 unfolds at Equator", *Microprocessor Report, 12(16)*, Dec. 1998

[16] T.C. Mowry, M.S. Lam and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching", in *Procs. of the 5th. Ann. Symp. on Programming Languages and Operating Systems (ASPLOS-V)*, pp.62-73, Oct. 1992

[17] E. Nystrom and A. E. Eichenberger, "Effective Cluster Assingment for Modulo Scheduling", in *Procs. of 31th. Int. Symp. on Microarchitecture*, pp. 103-114, 1998

[18] E. Özer, S. Banerjia and T.M. Conte, "Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures", in *Procs. of 31st Int. Symp. on Microarchitecture*, pp. 308-315, Nov. 1998

[19] S. Palacharla, N.P. Jouppi, and J.E. Smith, "Complexity-Effective Superscalar Processors", in *Procs. of the 24th. Int. Symp. on Computer Architecture*, pp. 1-13, June 1997

[20] B.R. Rau and C.D. Glaeser, "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing", in *Procs. on the 14th Ann. Workshop on Microprogramming*, pp. 183-198, Oct. 1981

[21] J. Sánchez and A. González, "Cache Sensitive Modulo Scheduling", in *Procs. of 30th. Int. Symp. on Microarchitecture*, pp. 338-348, Dec. 1997

[22] J. Sánchez and A. González, "The Effectiveness of Loop Unrolling for Modulo Scheduling in Clustered VLIW Architectures", in *Procs. of the 29th. Int. Conf. on Parallel Processing*, pp. 555-562, Aug. 2000

[23] Semiconductor Industry Association, "The National Technology Roadmap for Semiconductors: Technology Needs", 1997

[24] Texas Instruments Inc., "TMS320C62x/67x CPU and Instruction Set Reference Guide", 1998

[25] X. Vera, J. Llosa, A. González and C. Ciuraneta, "A Fast Implementation of Cache Miss Equations", in *Procs. of the 8th. Int. Workshop on Compilers for Parallel Computers*, pp. 319-326, Jan. 2000

[26] V.V. Zyuban, "Low-Power High-Performance Superscalar Architectures", *PhD Thesis, Dept. of Computer Science and Engineering, University of Notre Dame*, Jan. 2000