Register Aware Scheduling for Distributed Cache Clustered Architecture*

Zhong Wang
University of Notre Dame
Notre Dame IN 46556
e-mail: zwang1@cse.nd.edu

Xiaobo Sharon Hu
University of Notre Dame
Notre Dame IN 46556
shu@cse.nd.edu

Edwin H.-M. Sha
University of Texas at Dallas
Richardson TX 75083
edsha@utdallas.edu

Abstract— Increasing wire delays have become a serious problem for sophisticated VLSI designs. Clustered architecture offers a promising alternative to alleviate the problem. In the clustered architecture, the cache, register file and function units are all partitioned into clusters such that short CPU cycle time can be achieved. A key challenge is the arrangement of inter-cluster communication. In this paper, we present a novel algorithm for scheduling inter-cluster communication operations. Our algorithm achieves better register resource utilization than the previous methods. By judiciously putting the selected spilled variables into their corresponding consumer's local cache, the costly cross-cache transfer is minimized. Therefore, the distributed caches are used more efficiently and the register constraint can be satisfied without compromising the schedule performance. The experiments shows that our technique outperforms the existing cluster-oriented schedulers.

I. Introduction

Many high performance processors are designed with wide issue widths to exploit extensively instruction level parallelism (ILP). With the increased capability of overlapping operations comes the increased need to supply register bandwidth. Eventually the access time to and from a central register file becomes the bottleneck of the cycle time of the processor. It is very difficult to solve many problems associated with a large centralized register file, such as long access time, excess silicon area for address decoders, complicated bypassing logic, etc. Hence, a natural solution is to deploy a clustered architecture.

By distributing the registers, function units and cache into different clusters, register files can be located near their data consumers and producers, such that both access time and power are saved. The performance can also be greatly improved due to the reduced clock cycle time. Clustered architectures are gaining popularity, e.g., the architecture of Texas Instruments's *TMS*320*C*6000, Equator's *MAP*1000, Analog's *TigerShare* and HP *Lx* all adopt multiple clusters.

A major concern in clustered architecture is scheduling. In the clustered architecture, the inter-cluster MOVE operation is required whenever a non-local (not in the current cluster) variable is accessed. These inter-cluster data transfers may lead to undesirable increases in schedule latency if the previous centralized architecture scheduling algorithm is directly applied. An inefficient schedule often ruins the benefit obtained from the reduced processor cycle time. Therefore, an efficient scheduling technique which can cope with multiple clusters is very crucial to the success of such architecture.

*This work is supported in part by NSF under grant numbers MIP-9701416 and CCR02-08992.

In the context of multimedia and digital signal processing (DSP) applications, there exist a large number of loops. Loop schedule latency can significantly influence the whole system performance. Thus high quality software pipelining algorithms targeting VLIW clustered architecture are indispensable for such applications. It has been shown that rotation scheduling is one of the best software pipelining techniques in the centralized architecture [4]. Nevertheless, the traditional rotation algorithm does not consider the register constraint or cluster configuration, and thus limits the usage of this algorithm. This paper proposes an extended rotation scheduling algorithm to optimize the loop performance in the clustered architecture, where both register file and data cache are distributed among clusters. The proposed algorithm can efficiently explore the parallelism at the level of distributed function units and registers. It can handle arbitrary clustered datapath configuration and, along with schedule latency minimization, can effectively handle the register constraints.

One important aspect in clustered architecture is the consideration of inter-cluster communication. Previous methods [19, 1, 20] treat communication BUS as a type of hardware resource and inter-cluster communication operation as same as other operations during scheduling. Indeed, MOVE operation is different from other operations in that it has a significant impact on the register pressure of both provider and consumer clusters, while other operation only affect the local register pressure. Therefore, we collect all the MOVE operations (used to move operands between clusters) and schedule them at the later stage of the scheduler. With all the global information gathered, the MOVE operations can be scheduled in a more efficient way such that the register resource can be better utilized. The algorithm to schedule MOVE operations is presented and the register usage improvement can be seen from the experimental results.

Another important aspect in clustered architecture is the distributed cache. Distributed caches allow multiple clusters to perform memory operations simultaneously while a single cache only provides very limited parallel memory operations. In the contemporary architecture, the cache coherence is guaranteed by the hardware mechanism [7]. When the register resource is restrictive, some operands are needed to be spilled out to the local cache. The cross-cluster cache transfer will happen when the spilled operands are required by other clusters. The transfer between caches is rather costly which may degrade the system performance. Thus, cache allocation problem arises in order to reduce such costly cross-cluster transfer. None of the previous work has the consideration to associate the spill code insertion with distributed caches. By judiciously putting the spilled operand in the consumer's cluster local cache, we can substitute many cross-cache transfers with efficient cross-cluster register transfers and minimize the number of cross-cluster cache transfer. Hence the register constraint can be

satisfied without significantly increasing the schedule latency. Compared to the traditional schemes [20] which depend only on the hardware protocol to maintain the cache consistency, it is an obvious improvement to reduce the cross-cache transfer with the help of compiler.

The rest of paper is organized as follows. Section 2 reviews the necessary background knowledge and related previous research. Section 3 presents the framework of our scheduling algorithm and a detailed description. The experimental results and comparison are discussed in Section 4, and Section 5 concludes the paper.

II. Background

Our technique can be applied to the uniform nested loop which exists in a lot of DSP and multimedia applications. Moreover, more general linear index loops can be uniformized first, then use our technique to optimize the schedule. Detailed discussion on uniformization can be found in [15] and [21].

In a uniform nested loop, an *iteration* is the execution of the loop body once. It can be represented by a graph called *data flow graph* (DFG). A DFG is a directed weighted graph $G = (V, E, d, t)$ where V is the set of operation nodes, E is the edge set which defines the precedence relations among nodes in V, $d(e)$ is the data dependence for an edge $e \in E$ and $t(v)$ is the computation time of a node $v \in V$.

We briefly review the rotation scheduling algorithm, which will be referred to later. The *rotation scheduling algorithm* [4] is used to get a static compact schedule for one iteration. The inputs to the rotation scheduling algorithm are a DFG and its corresponding initial schedule. Rotation scheduling reduces the schedule length (the number of control steps needed to execute one iteration of the schedule) of the initial schedule by exploiting the concurrency across iterations. It accomplishes this by shifting the scope of the iteration in the initial schedule down so that nodes from different iterations appear in the same iteration scope. Intuitively speaking, this procedure is analogous to rotating tasks from the top of each iteration down to the bottom. Furthermore, this procedure is equivalent to retiming those tasks (nodes in the DFG) in which one delay can be deleted from all incoming edges and added to all outgoing edges, resulting in an intermediate retimed graph. Once the parallelism is revealed, the algorithm reassigns the rotated nodes to the earlier available positions so as to reduce the schedule length.

A. Architecture model

In this paper, we use the architecture model similar to that of HP *Lx* [7], as shown in Figure 1. It is a multi-cluster architecture. Each cluster is composed of a mix of register files and function units (each cluster may have a different configuration). Inter-cluster communication, achieved by explicit register-to-register move, is compiler-controlled and invisible to the programmer. To increase the parallelism and efficiency, each cluster has its own local data cache, thereby multiple data caches exist in the architecture. To establish main memory coherency in the presence of multiple memory access, A MESI-like (Modified, Exclusive, Shared, Invalid, which are four states that a cache line may be in) synchronization mechanism [3] can be used for multiple independent caches. This protocol is completely transparent to the ISA (Instruction Set Architecture). Both the coherence and the bus arbitration are managed by the hardware.

Regarding memory accesses, a load/store issued by a cluster first tries its local data cache. If the data is found, the access is satisfied with minimum latency. Otherwise, the access is solved by the MESI-like synchronization protocol. It will take much longer time to access data from other data cache (cross-cache transfer) or main memory.

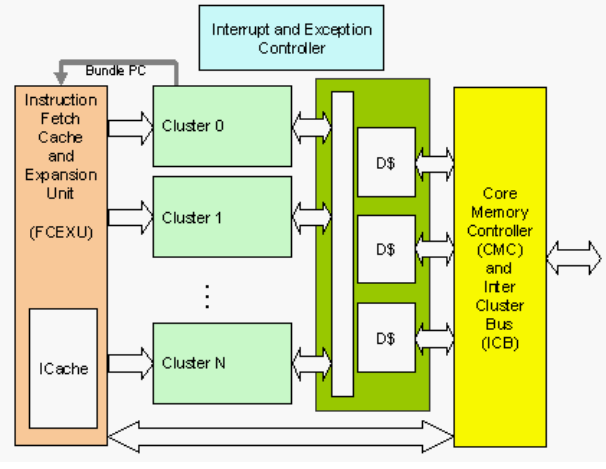

Fig. 1. Lx architecture model

B. Previous work

Previous research work related to cluster scheduling can be classified into two categories: scheduling with computational DAGs and scheduling with cyclic code. In [6], Desoli developed a two-phase binding algorithm called Partial Component Clustering. Ozer *et al* [13] presented a greedy binding /scheduling algorithm, which binds and schedules the ordered operations to a priority list of clusters in one step. The priority of the cluster is determined by several heuristics. Lapinskii and Jacome [11] proposed an algorithm whose initial schedule explores tradeoffs between in-cluster operation serialization and delays associated with inter-cluster data movement. An iterative scheme can be applied on this initial schedule to compact the schedule further. In [10], a code generation framework for Clustered ILP processors, which combines cluster assignment, register allocation and instruction scheduling, is presented. They use modified list scheduling algorithm to do cluster assignment and instruction scheduling. An on-the-fly register allocation is integrated into this algorithm. The above algorithms study scheduling of DAGs, and cannot be directly applied to the cyclic code. They can not take advantage of the inter-iteration data dependences, thereby lost many optimality.

Software pipelining [9] is efficient in scheduling cyclic code through moving operations among the iterations such that a shorter Initiation Interval (schedule length) can be achieved. Much work on software pipelining has been done in the centralized architecture, such as Modulo scheduling [17, 16], rotation scheduling [4] and retiming scheduling [2]. Here we briefly review algorithms that target clustered architecture. Sanchez *et al* [19] proposed the modulo scheduling algorithm which performs the cluster assignment and instruction scheduling in a single pass and considers possible improvements by loop unrolling. Akturan and Jacome [1] developed the CALIBeR framework for clustered embedded VLIW processors. The framework tries to optimize the schedule length as well as minimize the register usage and code size. It is the state-of-art cluster scheduling algorithm without considering distributed

cache. Another work dealing with clustered scheduling, which assumes a centralized cache, is presented in [23]. It takes into account of spill code insertion, which is the major improvement compared to previous work. One important difference between centralized and distributed cache lies in the cache allocation consideration, which is a key issue to reduce the cross-cache communication to improve the performance. To our best knowledge, the technique in [20] is the only cluster scheduling approach which considers both the distributed register file and caches. It uses Cache Miss Equations [8] to predict the influence of allocating memory operations and uses modulo scheduling to derive the compact schedule. The detail comparison in the experimental section show that our algorithm can achieve a better result.

III. Scheduling algorithm

The problem we are attacking in this paper is defined as follows:

Problem: *Given a DFG and a clustered datapath configuration, find a schedule that minimizes the schedule length and satisfies the register constraint. Insert spill codes if necessary to reduce the register requirement.*

From this definition, we can see that the solution to this problem can handle all major aspects of the clustered architecture, i.e., cluster configuration, register constraint, inter-cluster communication and distributed caches.

A. Scheduler framework

The framework of the scheduler is shown in Figure 2. The input includes the data flow graph and architecture specifications, i.e., clustered datapath configuration, cache configuration and register constraint. The output is a minimized feasible schedule for the given system.

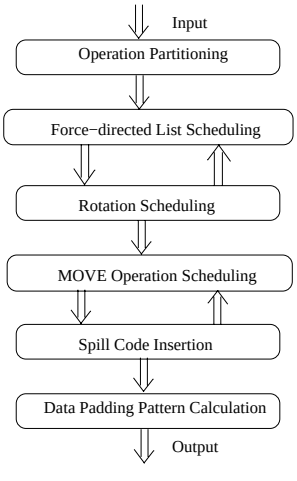

Fig. 2. The scheduling algorithm framework

The scheduler first partitions the DFG nodes into different clusters to achieve an efficient resource utilization. Then an initial schedule can be generated by the force-directed list scheduling module [14].[1] This initial schedule can be optimized by repeatedly applying rotation and partial rescheduling. Steps 2 and 3 are repeated several times until a minimal schedule length is reached.

[1] Any other scheduling technique, e.g, other kinds of list scheduling, can be used in this framework. We choose force-directed list scheduling because it can handle the resource constraint and operation concurrency effectively.

In the clustered architecture, rotation is performed on the basis of each cluster. That is to say, an operation is only rescheduled to another available control step in the same cluster. Only allowing operation rotation in the same cluster avoids the change of the inter-cluster move operations. With such a rotation scheme, the assignment of variables to clusters does not vary with rotation, thereby the communication workload on the communication bus remains constant with rotation. However, the variable lifetime may be changed by rotation.

With the compact schedule derived from the above steps, we can gather the global information of inter-cluster communication and iterate over the fourth and fifth steps to schedule the communication with the consideration of register constraint.

With the instruction rotation and spill code insertion, many operands from originally different iterations may be brought into one iteration and the data locality is increased. However, the likelihood of conflicts in the cache increases even if the cache capacity is enough. This problem may leads to the loss of data locality and the schedule improvement. Therefore, data padding [18] is applied in our framework as the last step in order to overcome this problem. By finding the suitable pad pattern, we can eliminate such cache conflict misses. The readers are refered to [18, 22] for the details.

B. Operation partitioning

As the first step, the DFG nodes are partitioned into different clusters. The objective is to minimize the communication cost and balance the load of function units in each cluster. A modified k-way partition algorithm is used in our technique for this purpose.

An important difference between scheduling acyclic and cyclic codes is reflected in this step's consideration. In literature, the k-way partition algorithm is seldom used in clustered scheduling because it does not consider the influence of the critical path on the schedule length. It is difficult to associate a good partition with a compact schedule length when scheduling acyclic code [6]. However, in cyclic code scheduling, most critical paths can be broken up by a suitable retiming. Hence a k-way partition algorithm can be used effectively in the first step to form a good initial partition. The following points should be kept in mind when implementing the operation partitioning step.

Point 1: Any edge between two nodes corresponds to an communication cost of 1 regardless of its weight, since the weight is only related to timing information.

Point 2: The balanced cluster load means that the load is evenly distributed to each cluster for every operation type, while the minimized communication cost is found from the global view.

Point 3: Strongly connected component (SCC) with small overall latency should be treated specially such that they are put into the same cluster with a high probability.

We use an example to help illustrate the last two points. Assume an architecture with two clusters, each of which has an ALU and a multiplication unit. For a DFG which has 2 multiplication (cost 3) nodes and 6 addition (cost 1) nodes, a good partition will allocate 3 additions and 1 multiplication to each cluster. A partition, with 2 multiplications in one cluster while 6 additions in the other cluster, is not acceptable though the cluster cost is balanced. As of the combination of 1 multiplication and 3 additions, it should be decided by taking into consideration all the communication cost between any two nodes

to minimize the communication cost. For point 3, a SCC of a DFG is a subgraph such that for every pair of nodes n_i and n_j belonging to this SCC, there exists a path $n_i \rightarrow n_j$ and $n_j \rightarrow n_i$. A SCC with small overall latency always exerts an extra restriction for the schedule. Retiming cannot change the overall latency of such a cycle. If a critical path exists in this kind of SCC, it cannot be broken up by retiming. Therefore, it is beneficial to put such SCC in one cluster.

K-way partition is an NP-complete problem and many heuristics algorithms exist. We modified the algorithm in [12] due to its easy implementation and fast speed. Point 2 is handled by artificially assigning special costs to different kinds of operations. We take care of Point 3 by a post-processing step after the DFG is partitioned. Although the k-way partition algorithm we used is fairly simple, it is good enough to generate a partition from which an efficient schedule can be derived. However, with a better k-way partition heuristic algorithm which can take the above three points into account, some improvements of the final schedule may be expected.

C. Move operation scheduling

When an operation requests a variable in another cluster, an inter-cluster MOVE operation is needed. The schedule of MOVE operations influences the maximum register requirement significantly. Through a better MOVE operation schedule, register loads may be distributed to clusters more evenly such that the overall register requirement is reduced. By considering the MOVE operation scheduling after rotation, we have a comprehensive view of the global register usage and can use it to achieve more efficient register usage. The global MOVE scheduling algorithm is shown in Algorithm 1.

Algorithm 1 MOVE operation scheduling algorithm

Input: MOVE operations in list L and the schedule after rotation.
Output: MOVE operation schedule which can generate the efficient register usage
1. Derive the register usage map for cluster part.
2. Sort L in the increasing order of lifetime.
WHILE (L is not empty) **do**
  Pop out the head h of L
  FOREACH (control step i in h's lifetime) **do**
    Calculate $F(i)$. Push it into list $Force\_list(h)$
  ENDFOR
  Sort $Force\_list(h)$ in the increasing order.
  Tentatively schedule h in the control step Con corresponding to the least force in $Force\_list(h)$.
  WHILE (BUS confliction exist between h and a previously scheduled OLD) **do**
    if OLD's lifetime $\in$ h's lifetime **then**
      Tentatively schedule h in the control step Con' with the second least force.
    else
      Select h as the instruction with the less second least force of two instructions, and schedule it to the corresponding control step.
    end if
  ENDWHILE
  Update the move part register usage map
ENDWHILE

The register requirement of a schedule can be derived by a register usage map. In our algorithm, we divide the register usage map into two parts, *cluster part* for all except MOVE operations, and *move part* for only MOVE operations. In the move part, the lifetime of a MOVE operation starts from the last control step when the variable to be moved is alive in the producer cluster and ends at the first control step when the variable is needed in the consumer cluster.[2] The MOVE operation can be scheduled at any control step belonging to its lifetime. Scheduling a MOVE at different control steps will have different impact on the register usage map of the move part. For example, scheduling a MOVE operation whose lifetime is $1 \rightarrow 5$ at control step 2 will add 1 to the move part register usage map of the producer and consumer clusters at control steps $1,2$ and $2,3,4,5$, respectively, while scheduling it at control step 4 will add 1 at control steps $1,2,3,4$ and $4,5$, respectively. If the maximum register requirement of the *cluster part* occurs at the control step 3 in the consumer cluster, we prefer to scheduling this MOVE at control step 4 since it will not increase the overall register requirement.

Suppose the largest register requirement in the *cluster part* is *MAX*. If a MOVE operation is scheduled at a certain step *Con*, a new move part register usage map can be derived. In each control step *i*, we find the maximum register requirement value M_i for all clusters at this control step from the overall register usage map. The *force* of scheduling MOVE at *Con* at this control step is defined as $F(Con)_i = M_i - MAX$, and the *force of scheduling MOVE at Con* is defined as $F(Con)$, which is the largest of $F(Con)_i$. We always try to schedule the MOVE operation to a control step b which has the least force $F(b)$, where b belong to the MOVE operation's lifetime. The objective is to leave more scheduling space for other MOVE instructions.

In the algorithm, all the MOVE instructions are considered in the order of increasing lifetime. The observation behind this considering order is that the instruction with larger lifetime has more scheduling freedom. Because of the BUS constraint, a MOVE instruction may not be able to be scheduled at its optimal control step. One of two conflict instructions will be scheduled at a suboptimal control step. The selection of such an instruction depends on the lifetime relationship and the comparison of the suboptimal force values. It is easy to verify that the time complexity of this algorithm is $O(m^2n^2)$, where m is the schedule length and n is the number of MOVE operations.

D. Spill code insertion

If the register requirement still exceeds the constraint after the first four steps in Figure 5, spill codes need to be inserted to satisfy the register constraint. Spill code insertion has been considered in many papers [23, 5]. None of them have dealt with distributed cache. On the other hand, we believe spill code insertion should be applied to the **entire** schedule instead of the partial schedule as in [23]. With such scheme, we can easily identify those operands with longest lifetime, whose spill can reduce register pressure to the largest extent.

In the clustered architecture with distributed cache, the consideration of spill code insertion includes two aspects: which variable is spilled (to free more registers) and which cache should the spilled variable be put in (to reduce the costly cross-cache transfer).

Spill code is added through a pair of memory operations: writing back to the local cache and loading it later. Each cluster has its own local caches, which can be accessed much faster than main memory and remote caches. To minimize the cross-cache transfer, a spilled variable is always written back to the local cache of its **consumer**, i.e., the variable has been moved to the desired cluster before spilled out to the cache. By this way, the costly cross-cache transfer is replaced with cross-cluster register MOVE, which has much higher efficiency.

If a cluster demands more registers than available, several steps are carried out to get rid of the violation.

1 Find a variable with the longest lifetime and its spilling can help reduce the register pressure.

[2] In the case of two MOVE operations deal with the same moving variable but have different control steps, a and b in the consumer cluster, the MOVE operation with the later control step can be omitted, and the span of lifetime $b-a$ (assuming $b \geq a$) is counted into the cluster part register usage map.

2 If the variable's lifetime is longer than the overall cost of writing back and loading from the local cache, the corresponding spill code is inserted.

3 Otherwise, find the control step with the largest register requirement. Distribute the operations in this control step into two control steps to reduce the register pressure, thereby increase the schedule length by one control step. Because of the change of schedule length, the MOVE operations are rescheduled (correspondes to the iterative execution of steps 4 and 5 in Figure 2.

4 The above steps are repeated until all the register constraints are satisfied.

These steps are effective in reducing the register pressure since the general register constraint violation is often caused by long variable lifetimes that span several iterations. It is advantageous to swap them out of the register file to decrease the register usage. If no such long lifetime variable exists and the register limit is still surpassed, the register resource is really tight. The only way to satisfy the constraint is to increase the schedule length to provide more room for scheduling, such that the register pressure is alleviated by reducing the number of living variables at the same time.

IV. Experimental results

The effectiveness of our technique is illustrated by running a set of benchmarks from [1]. We compared our results with CALiBeR in [1] and Modulo scheduling in [19]. The experimental results for the latter two methods are extracted from [1]. In order to make a fair comparison, we use the same set of benchmarks with exactly the same DFGs, and the same cluster configuration as [1]. Table 1 lists the benchmarks and the clustered datapath configuration. In this table, the corresponding benchmarks from left to right are *Lattice Filter, 2 Cascaded Biquad Filter, Avenhous Filter, 4 Cascaded FIR Filter, AR Filter, 4 Cascaded Biquad Filter, DCT-DIT*. The number of nodes in the DFG is listed after the benchmark's name. The clustered datapath is specified in the form of number of ALU (a), multiplication (m) and load/store (x) units.

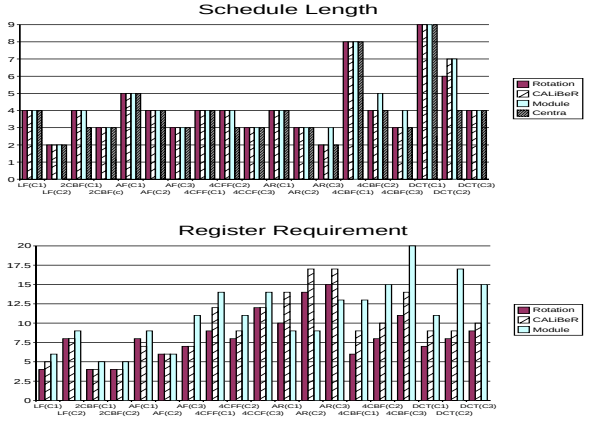

Fig. 3. Schedule length and register requirement without register constraint

In our experiment, the minimal schedules for all benchmarks can be found in one minute on a SUN Ultra-2 SPARC workstation, which demonstrates the time efficiency of our technique. In order to evaluate the efficiency of the scheduler itself, we show the experimental results in Figure 3. The schedule length for 4 different schedulers are listed. They are *register aware scheduling* (rotation), *CALiBeR* from [1] (CALiBeR), modulo scheduling in [19] (modulo), and the RS-FDRA in [2] (Centra). RS-FDRA is a centralized architecture software pipelining scheduling algorithm. It is used to provide a criteria to show how well the other three schedulers can perform in clustered architecture. It has been shown in [2] that this algorithm and rotation scheduling can always achieve the best results in the centralized architecture. In a clustered architecture, after the function units are partitioned into clusters, the schedule length cannot get better due to the extra cost introduced by inter-cluster communication. Therefore, the results from RS-FDRA can be regarded as the lower bound for other algorithms. CALiBeR is selected for comparison because it is the start-of-art software-pipelining algorithm in clustered architecture. The modulo scheduling in [19] is selected for comparison because it is the baseline scheduling algorithm of [20]. In Figure 3, the result of register requirement for RS-FDRA is not shown, because it is meaningless to compare the register requirement for different architectures.

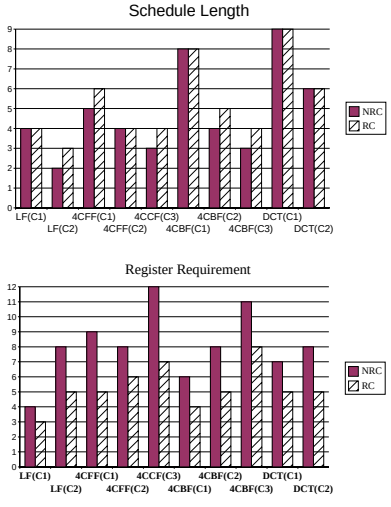

Fig. 4. The schedule length and register requirement if register resource is restricted. NRC and RC represent the performance without register constraint and with register constraint, respectively.

If the register resource is not restricted, we can see from Figure 3 that all three clustered scheduling algorithms are quite effective in that they can reach the lower bound in most cases. However, from the pointer view of register cost, as shown in Figure 3, our technique requires the least number of registers than the other two schedulers. The register usage comparison demonstrates that the global MOVE operation schedule can indeed reduce register cost.

When the register resource is restrictive, spill codes have to be inserted to swap some variables out of the register file. CALiBeR does not consider the register constraint, thereby cannot obtain a schedule under the rather restrictive register constraint. Through judiciously inserting spill code, our technique can reduce the register requirement by 1/3 -1/2 with almost the same schedule length, as shown in Figure 4. Moreover, our technique can still get a reasonable result if the register constraint becomes tighter. In such a case, the schedule length is expected to become longer to modify the register requirement distribution and accommodate the swapping time between the register file and local cache.

As we mentioned before, the algorithm in [20] is the only existing algorithm which can handle the distributed cache. It

	LF (16)		2CBF (16)		AF (20)			4CFF (32)			AR (34)			4CBF (32)			DCT (48)		
	C1	C2	C1	C2	C1	C2	C3	C1	C2	C3	C1	C2	C3	C1	C2	C3	C1	C2	C3
Cluster1	1a 1m	2a 2m	1a 1m	2a 2m	1a 1m 1x	1a 1m 1x	2a 2m 1x	2a 2m	2a 2m	3a 3m	2a 2m 2x	3a 3m 2x	3a 4m 2x	1a 1m	2a 2m	4a 4m	2a 1m	3a 1m	3a 1m
Cluster2	1a 1m	2a 2m	1a 1m	2a 2m	1a 1m 1x	1a 1m 1x	2a 2m 1x	2a 2m	2a 2m	3a 3m	2a 2m 2x	3a 3m 2x	3a 4m 2x	1a 1m	2a 2m	4a 4m	2a 1m	3a 1m	3a 1m
Cluster3						1a 1m 1x			2a 2m									3a 1m	3a 1m
BUS	1	1	1	1	1	2	2	1	2	1	2	2	3	1	1	1	2	3	5

TABLE I
BENCHMARKS AND THE CLUSTERED CONFIGURATION

takes the distributed cache into consideration by integrating a Cache Miss Equation (CME) [8] solver into the baseline algorithm [19]. Solving CME is NP-complete. By deploying some heuristics, they try to solve CME and find a schedule to reduce the cache conflict misses. Because the baseline algorithm does not consider the cache conflict, the schedule length obtained from the baseline algorithm can be regarded as the lower bound of their algorithm. Although this CME solver can improve the cache performance to some extents, it cannot eliminate cache conflict misses. The existence of such misses may severely compromise the schedule. As shown in Figure 3, our scheduler outperforms the baseline algorithm in in both schedule length and register requirement. Moreover in our algorithm, cache conflict misses are eliminated through data padding. Therefore, We believe that our scheduler is better than the algorithm in [20].

V. CONCLUSION

We have presented a register aware scheduling framework suitable for compilers targeting clustered VLIW processors with distributed cache. A novel global inter-cluster communication scheduling algorithm is proposed to efficiently utilize the register resource. To satisfy the restrictive register constraint, spill codes are wisely inserted. The proposed algorithm can handle arbitrary clustered configuration and, along with latency minimization, can effectively handle register constraint. The experimental results demonstrate that our technique is superior to the existing algorithms.

REFERENCES

[1] C. Akturan and M. Jacome. Caliber: a software pipelining algorithm for clustered embedded vliw processors. In *Proceeding of IEEE/ACM International Conference on Computer Aided Design*, pages 112–118, 2001.

[2] C. Akturan and M. Jacome. Rs-fdra: A register sensitive software pipelining algorithm for embedded vliw processors. In *Proceeding of 9th International Symposium on Hardware/Software Codesign*, April 2001.

[3] J. Archibald and J. L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4), November 1986.

[4] L.-F. Chao, A. LaPaugh, and E. H.-M. Sha. Rotation scheduling: A loop pipelining algorithm. *IEEE Transactions on Computer Aided Design*, 16(3), March 1997.

[5] A. K. Dani, V. J. Ramanan, and R. Govindarajan. Register-sensitive software pipelining. In *Proc. Merged 12th Intl. Parallel Processing and 9th Intl. Symposium on Parallel and Distributed System*, April 1998.

[6] G. Desoli. Instruction assignment for clustered vliw dsp compilers: A new approach. Technical Report HPL-98-13, Hewlett-Packard Company, 1998.

[7] P. Faraboschi, G. Brown, and J. A. Fisher. Lx: A technology platform for customizable vliw embedded processing. In *Proc. of 27th Annual International Symposium on Computer Architecture*, Vancouver, Canada, June 2000.

[8] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: an analytical representation of cache misses. In *Proceeding of International Conference on Supercomputing*, pages 317–324, July 1997.

[9] R. Jones and V. Allan. Software pipelining: An evaluation of enhanced pipelining. In *Proc. International workshop on Microprogramming and Microarchitecture*, pages 82–92, Nov 1991.

[10] K. Kailas, K. Ebcioğlu, and A. Agrawala. Cars: A new code generation framework for clustered ilp processors. In *Proc. International Symposium on High Performance Computer Architecture*, pages 133–143, Monterrey, Mexico, Jan 2001.

[11] V. Lapinskii and G. Jacome, M.F.and de Veciana. High-quality operation binding for clustered vliw datapaths. In *Proceeding of Design Automation Conference*, pages 702–707, 2001.

[12] C. Lee, M. Kim, and C. I. Park. An efficient k-way graph partitioning algorithm for task allocation in parallel computing systems. In *Proceeding of the First International Conference on Systems Integration*, pages 748–751, 1990.

[13] E. Ozer, S. Banerjia, , and T. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proceeding of the 31th Annual Intern. Symposium on Microarchitectures,*, pages 308–315, 1998.

[14] P. G. Paulin and J. P. Knight. Force directed scheduling for the behavioral synthesis of asics. *IEEE Trans. on Computer Aided Design*, 8(1), June 1989.

[15] P. Quinton and V. V. Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1, 1998.

[16] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, Nov 1994.

[17] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc. 14th Annual Workshop on Microprogramming*, pages 183–198, 1981.

[18] G. Rivera and C. W. Tseng. Data transformation for eliminating conflict misses. In *Proceedings of the ACM SIGPLAN'98 conference on Programming Language Design and Implementation*, pages 38–49, 1998.

[19] J. Sanchez and A. Gonzalez. Instruction scheduling for clustered vliw architectures. In *Proceeding of the 13th International Symposium on System Synthesis*, pages 41–46, 2000.

[20] J. Sanchez and A. Gonzalez. Modulo scheduling for a fully-distributed clustered vliw architecture. In *Proceeding of 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 124–133, 2000.

[21] W. Shang, E. Hodzic, and Z. Chen. On uniformization of affine dependence algorithms. *IEEE Transactions on Computers*, 45(7), 1996.

[22] Z. Wang, E. H.-M. Sha, and X. Hu. Combining partitioning and data padding for scheduling multiple loop nests. In *Proc. International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 67–75, Atlanta, GA, Nov 2001.

[23] J. Zalamea, J. Llosa, E. Ayguade, and M. Valero. Modulo scheduling with integrated register spilling for clustered vliw architecture. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 160–169, Austin, Texas, December 2001.