A Register File Architecture and Compilation Scheme for Clustered ILP Processors*

Krishnan Kailas[1], Manoj Franklin[2], and Kemal Ebcioğlu[1]

[1] IBM T. J. Watson Research Center, Yorktown Heights, NY, U.S.A.
{krish, kemal}@watson.ibm.com
[2] Department of ECE, University of Maryland, College Park, MD, U.S.A.
manoj@eng.umd.edu

Abstract. In Clustered Instruction-level Parallel (ILP) processors, the function units are partitioned and resources such as register file and cache are either partitioned or replicated and then grouped together into on-chip clusters. We present a novel partitioned register file architecture for clustered ILP processors which exploits the temporal locality of references to remote registers in a cluster and combines multiple inter-cluster communication operations into a single broadcast operation using a new `sendb` instruction. Our scheme makes use of a small *Caching Register Buffer* (CRB) attached to the traditional partitioned local register file, which is used to store copies of remote registers. We present an efficient code generation algorithm to schedule `sendb` operations on-the-fly. Detailed experimental results show that a windowed CRB with just 4 entries provides the same performance as that of a partitioned register file with infinite non-architected register space for keeping remote registers.

1 Introduction

Registers are the primary means of inter-operation communication and inter-operation dependency specification in an ILP processor. In contemporary processors, as the issue width increase, additional register file ports are required to cater to the large number of function units. Further, a large number of registers are required for exploiting many aggressive ILP compilation techniques, as well as for reducing the memory traffic due to spilling. Unfortunately, the area, access delay, and power dissipation of the register file grows rapidly with the number of ALUs [1]. Clearly, huge monolithic register files with large number of ports are either impractical to build or limit the cycle time of the processor. The partitioned and replicated register file structures used in clustered ILP processors [1–8] are two promising approaches to effectively address the issues related to large number of ports, area and power of register files. A typical cluster consists of a set of function units and a local register file, as shown in Fig. 1. A local register file can provide faster access times than its monolithic counter-part, because each local register file has to cater only to a subset of all the function units in the processor. If the register file is *replicated* (each local register file shares the entire architected name space), then copy operations are required to maintain the coherency among the local register files. On the other hand, if the

* Supported in part by U.S. National Science Foundation (NSF) through a CAREER grant (MIP 9702569) and a regular grant (CCR 0073582).

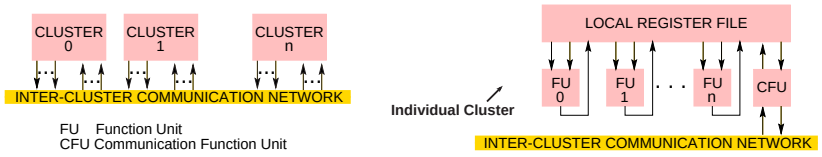

Fig. 1. Generic Clustered ILP Processor Model

register file is *partitioned* (each local register file has a non-intersecting subset of the architected name space), then copy operations are needed to access registers from remote clusters. The main advantage of a partitioned register file over a replicated register file is that it can reduce the number of write ports as well as the size of the local register file, thereby providing shorter access delays, smaller area and lower power requirements. In this paper, we shall concentrate on partitioned register file architectures for statically scheduled processors.

In general, the partitioned register file schemes used in statically scheduled clustered ILP processors belong to one of the following two categories:

- Registers are copied to remote clusters ahead of their use to hide the inter-cluster communication delay (eg. Multiflow Trace [9], M-machine [10], and Limited Connectivity VLIW [11]). Often several copies of a register are maintained in the local register files of several clusters, increasing the register pressure due to inefficient use of architected name space in this scheme.
- Remote registers are requested either on demand (eg. TI's clustered DSPs[1] [4]) or using send-receive operations (eg. Lx [2]), each time they are referenced. The potential drawback of this approach is that it demands more inter-cluster communication bandwidth, and hence may possibly delay the earliest time a remote register can be used when the interconnect is saturated.

In this paper, we propose a new partitioned register file with a *Caching Register Buffer (CRB)* structure, which tries to overcome the above drawbacks. The CRB is explicitly managed by the compiler using a single new primitive instruction called `sendb`.

The rest of this paper is organized as follows. Section 2 describes our CRB-based partitioned register file scheme, and discuss some related programming and hardware complexity issues. In section 3, we briefly discuss the code generation framework used and present an on-the-fly scheduling algorithm to explicitly manage the CRB using the new `sendb` operations. The experimental evaluation of the proposed CRB-based partitioned register scheme is discussed in section 4. We present related work in section 5, followed by conclusion in section 6.

2 Partitioned register file with CRB

The basic idea behind our scheme is to use compile-time techniques to exploit the temporal locality of references to remote registers in a cluster. Instead of allocating and copying a remote register value to a local register (and thereby wasting an architected register name), the remote register contents are stored in a *Caching Register Buffer* (CRB), which is a fast local buffer for caching the

[1] In addition, TMS320C6x series DSPs also support `rcopy` OPs described in section 4.

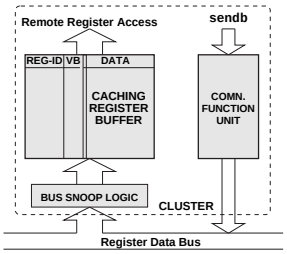

Fig. 2. Caching Register Buffer and Communication Function Unit

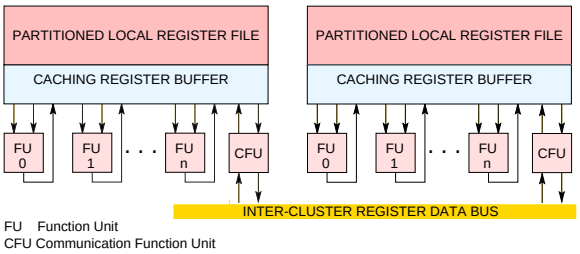

Fig. 3. A 2-cluster processor with CRB-based partitioned register file

frequently accessed remote registers ahead of their use. The CRB is therefore functionally similar to a small fully-associative cache memory, and it may be implemented using an array of registers. There is a tag field (REG-ID) and a one bit flag (VALID-BIT) associated with each CRB data entry as shown in Fig. 2. The tag (REG-ID) contains the unique address (architected name) of the cached remote register, and the VALID-BIT is used for validating the data and tag.

An operation trying to access a remote register first performs an associative search in the REG-ID array. If a valid copy of the remote register being accessed is available in the CRB, then the data value is returned from the appropriate CRB data register (akin to a cache hit). A hardware-initiated *send-stall* bubble is injected into the pipeline automatically if a valid copy of the requested remote register is not found in the CRB. During the send-stall cycle, which is similar to a cache miss, the data is loaded from the remote register file and the execution of the pipe is resumed.

Any local register can be sent to the CRBs of remote clusters via an explicit *send-broadcast* (`sendb`) operation using a *communication function unit* (CFU). The CRBs and the CFUs of all clusters are interconnected via one or more *register data buses* as shown in Fig. 3. All the resources involved in inter-cluster communication — CRBs, CFUs, and register data buses — are reserved on a cycle-by-cycle basis at compile-time. The `sendb` operation (OP) is scheduled on the CFU by the compiler such that the remote register is cached in the CRB of a remote cluster ahead of its use. The `sendb` OP has the following format:

`sendb`	`SrcRegID`	`ClusterIDbitvector`

where `SrcRegID` is the ID of the register that has to be sent to other clusters, and `ClusterIDbitvector` is a bit vector with bit positions indicating the remote cluster IDs to which the register value is to be sent. The width of the `ClusterIDbitvector` is therefore equal to the number of clusters in the processor. The CRB's bus interface unit snoops the register data bus; if the bit position in `ClusterIDbitvector` matches its cluster ID then it updates the appropriate CRB data register, concurrently initializing its tag REG-ID entry and setting its VALID-BIT to 1.

2.1 Precise exceptions and calling conventions

An important issue that needs to be addressed is how to support precise exceptions in the presence of a CRB. Only the contents of the architected registers

(the local partitioned register file) need to be saved on an interrupt by the interrupt handler code. In order to make the CRB structure transparent to the user program, the interrupt return (`rfi`) OP will invalidate the contents of all CRBs by resetting all VALID-BITs to 0. Therefore, after returning from the interrupt handler, if a valid copy of the requested remote register is not found in the CRB, it is loaded directly from the register file during send-stall cycles. An alternative approach is to save and restore the entire CRB image on an interrupt. However, this approach may result in large interrupt latencies and slower context switches.

In addition to physical partitioning, there is also a *logical partitioning* of register name space due to calling conventions. The logical partitioning due to calling conventions should co-exist with the physical partitioning in the new partitioned register file with CRB as well. We use a partitioning scheme in which both caller-save and callee-save registers are partitioned and distributed equally among all clusters [8]. However, the logical partitioning of register name space and the calling convention provide only a protocol to preserve the callee-save registers across function calls. This brings up the important issue of dealing with the changes in the cached callee-save registers in the CRBs across function calls. Note that, for the correct execution of the program, the data in any valid CRB register and the architected register specified by its REG-ID tag should be the same. A simple solution to this problem is to invalidate the CRB contents upon return from a function call and load a copy of the register directly into the CRB and function unit. Such send-stall cycles can be completely eliminated by scheduling a `sendb` OP after the `call` OP for each one of the registers that are live across the function call. Our preliminary experiments showed that such schemes can either increase the number of send-stall cycles or increase the code size as well as schedule length, especially for function-call-intensive benchmark programs. This led us to the following hardware-based solution. The basic idea is to use "windowed" CRBs, which allow saving different versions of the CRB contents in different register windows. Like any register windowing scheme, such as SUN PicoJava processor's "dribbling" mechanism, each call nesting level selects a different CRB window automatically. When the nesting level of calls exceeds the maximum number of hardware CRB windows in the processor, the contents of the oldest CRB window are flushed to cache.

The number of ports in a typical CRB structure is less than the number of ports in a partitioned register file [8]. Because we need only a 4-way associative search logic for good performance (based on our simulation results presented in section 4), one can argue that it is feasible to build a windowed CRB structure that has the same access time of the partitioned register file attached to it. However, CRB may need a larger area than a traditional register file.

3 Compilation scheme

3.1 Overview of code generation framework

We used the CARS code generation framework [12, 8] which combines the cluster assignment, register allocation, and instruction scheduling phases to generate efficient code for clustered ILP processors. The input to the code generator is

a dependence flow graph (DFG) [13] with nodes representing OPs and directed edges representing data/control flow. The basic scheduling unit can be a set of OPs created using any "region" formation or basic block grouping heuristics.

During code generation, CARS dynamically partitions the OPs in the scheduling region into a set of mutually exclusive aggregates (groups) — `unscheduled`, `ready list`, and `vliws`. The data ready nodes in the `unscheduled` aggregate are identified and moved into the `ready list`. The nodes in the `ready list` are selected based on some heuristics for combined cluster assignment, register allocation and instruction scheduling (henceforth referred to as CARS*cheduling*). CARS also performs on-the-fly global register allocation using usage counts during CARScheduling. This process is repeated until all the nodes of the DFG are CARScheduled and moved to the appropriate `vliws`.

The CARS algorithm extends the list-scheduling algorithm [14]. In order to find the best cluster to schedule an OP selected from the `ready list`, the algorithm first computes the resource-constrained *schedule_cycle* in which the OP can be scheduled in each cluster, and then the minimum value of *schedule_cycle* as *earliest_cycle*. Based on the *earliest_cycle* computed, the OP is either scheduled in the current cycle on one of the clusters corresponding to *earliest_cycle* or pushed back into the `ready list`. Often inter-cluster copy OPs have to be inserted in the DFG and retroactively scheduled in order to access operands residing in remote clusters. An operation-driven version of the CARS algorithm is used for this purpose. In the list-scheduling mode, only the most recently scheduled `vliws` aggregate's resource availability is searched to make a cluster-scheduling decision. On the other hand, in the operation driven scheduling mode[2], resource availability in a set of `vliws` aggregates are considered to make the cluster-scheduling decision. In order to find the best VLIW to schedule the inter-cluster copy OP, the algorithm searches all the `vliws` aggregates starting from the Def cycle of the operand (or from the cycle in which the join node of the current region is scheduled, if operand is not defined in the current region) to the current cycle. In the next section, we describe how we have extended the CARS algorithm for scheduling `sendb` operations on-the-fly.

3.2 On-the-fly scheduling of `sendb` operations

Our basic approach is to schedule a `sendb` OP whenever it is beneficial to schedule a node on a cluster in which one or both of its remote source registers are not cached in the CRB and the inter-cluster communication bus is available for broadcasting the register value. We also want to combine multiple inter-cluster copy operations into a single `sendb` OP on-the-fly while CARScheduling nodes in the DFG in topological order, without back-tracking. In order to combine multiple inter-cluster copy OPs, we keep track of `sendb` OPs scheduled for accessing each live register. The information is maintained by the code generator as a list of `sendb` OPs inside each live physical register's resource structure.

Our scheme may be explained using an example of scheduling nodes in a basic block. Assume that a `sendb` OP has already been scheduled for accessing

[2] Due to the large search space involved with operation-driven scheduling, use of this mode has been restricted to scheduling inter-cluster copy OPs and spill store OPs.

Algorithm 1 Algorithm for CARScheduling `sendb` OPs

SCHEDULE-SENDB($\mathcal{O}p$, $RegId$, $ClusterID$)
/* $\mathcal{O}p$ is an OP trying to access remote register≡$RegId$ in cluster≡$ClusterID$ */
1: find `LifeInfo` of the life to which register $RegId$ is currently assigned
{`LifeInfo` is a list of Defs in the web of Def-Use chain corresponding to a life (live range) in DFG}
2: find the subset of Defs $SrcDefs \in$ `LifeInfo` that can be reached from $\mathcal{O}p$
3: **for** each Def $\mathcal{D}$ in $SrcDefs$ **do**
4: **if** no `sendb` OP is scheduled for $\mathcal{D}$ **then**
5: schedule a new `sendb` OP for $\mathcal{D}$ using OPERATION-CARS in the current block
6: return
7: **end if**
8: **end for**
9: **for** each Def $\mathcal{D}$ in $SrcDefs$ **do**
10: find the `sendb` OP $\mathcal{S}$ scheduled for $\mathcal{D}$
11: set the bit corresponding to $ClusterID$ in `ClusterIDbitvector` of $\mathcal{S}$
12: **end for**

a remote register, say `r10`, from a cluster C_1, and that we want to use the same register `r10` in a different cluster C_2 at some time later while scheduling a different Use-node of the Def mapped to `r10`. Our algorithm first checks if there are any `sendb` OPs scheduled for register `r10`. As there is one such `sendb` OP scheduled already for `r10`, we simply set the bit corresponding to the new destination cluster C_2 in the `ClusterIDbitvector` operand of the `sendb` OP (instead of scheduling a new inter-cluster operation as in traditional schemes). The process is repeated whenever a Use-node of the Def mapped to `r10` needs the register in a different cluster. Therefore, by the end of scheduling all the OPs in DFG, the algorithm would have combined all the inter-cluster copy OPs into the single `sendb` OP by incrementally updating the `ClusterIDbitvector` operand of the `sendb` OP.

The scheme discussed above for scheduling `sendb` OPs in a basic block can be extended, as shown in Algorithm 1, to the general case of scheduling `sendb` OPs for accessing a remote register assigned to a web of Def-Use chains (corresponding to a life that spans multiple basic blocks). The nodes that use such registers that are allocated to webs of Def-Use chains may have multiple reaching definitions. Clearly, there are two options: either schedule a `sendb` OP for each one of the Def nodes that can be reached from the Use node, or schedule one `sendb` OP before the Use node in the current basic block. Our preliminary experiments showed that former is attractive only when there are empty issue slots in already scheduled VLIWs (low ILP applications), because if there are no issue slots then we have to open a new VLIW just for scheduling a `sendb` OP. Moreover, if there are not many remote uses for the register, then the latter option is better than scheduling multiple `sendb` OPs. In view of these observations, our algorithm uses the following scheme to reduce the number of `sendb` OPs. If `sendb` OPs have already been scheduled for all the reaching definitions corresponding to a

Use, then we update the `ClusterIDbitvector` of all those `sendb` OPs (lines 9-12). Otherwise, we schedule a `sendb` OP in the current basic block using the operation-driven version of CARS algorithm (line 5) [8].

4 Experimental results

We used the chameleon VLIW research compiler [15] (gcc version) with a new back-end based on the CARS code generation framework [12] for the experimental evaluation of our new partitioned register file scheme. We have developed a cycle-accurate simulator to model the clustered ILP processor with CRB and inter-cluster buses. We use the *compiled simulation* approach: each VLIW instruction is instrumented and translated into PowerPC assembly code that calls the CRB simulator. For each VLIW instruction, the simulator processes all the local and remote register file access requests, updates the CRB contents and keeps track of the number of CRB hits and misses based on whether the requested register was found in the CRB or not. The send-stall cycles are computed based on the assumption that multiple CRB misses are serviced serially subject to the availability of inter-cluster buses. FIFO replacement policy is used in the CRB.

We compared the CRB-`sendb` scheme with the traditional partitioned register file scheme using inter-cluster copy (`rcopy`) OPs. We generated executable compiled simulation binaries for the 14 benchmark programs used (8 from SPEC CINT95, 2 from SPEC CINT2000, and 4 from MediaBench). The binaries are run to completion with the respective input data sets of the benchmark programs. We used the execution time of benchmark programs in cycles as a metric to compare the performance of 4 different clustered ILP processors listed in Table 1. Infinite cache models were used for all configurations and the penalty for flushing/restoring a CRB window is assumed to be 5 cycles. The `rcopy` OPs are semantically equivalent to register `copy` OPs; however their source and destination registers reside in different clusters. For each remote register access, an `rcopy` OP is scheduled on the destination cluster to copy a register from a remote register file to an *infinitely* large non-architected name space in the local register file. The rationale is to compare the performance of CRB-`sendb` scheme with the (unrealistic) upper-bound on performance that may be achieved via traditional partitioned register file scheme using `rcopy` OPs. We studied three partitioned register file configurations with CRB sizes 4, 8 and 16.

The clustered machines are configured such that the issue width and resources of the base machine are evenly divided and assigned to each cluster as shown in Table 1. Condition code registers (CCRs) and buses are treated as shared resources. The function units, which are fully pipelined, have the following latencies: Fix, Branch, and Communication: 1 cycle; Load and FP: 2 cycles; FP divide and sqrt: 9 cycles. Figures 4, 5, 6 and 7 show the speed up (ratio of execution times) with respect to the base single cluster processor for the four different 8 ALU clustered processors for all the benchmarks studied. In 2-cluster configurations, CRB-`sendb` scheme does not provide any significant performance improvement for most of the benchmarks. This is mainly because, in a 2-cluster machine, the `sendb` OP degenerates into an `rcopy` OP because there is only one target cluster. However, in the 4-cluster configurations, the CRB-`sendb` scheme

Table 1. Configurations of different 8 ALU processors with partitioned register files studied. [1] Each CRB has 4 windows.

Configuration		Local resources per cluster				Global resources	
		ALUs	GPRs	FPRs	CRB/buf[1]	CCRs	Buses
single cluster (base)		8	64	64	-	16	
`rcopy`	1 bus 4 clusters	2	16	16	∞	16	1
	1 bus 2 clusters	4	32	32	∞	16	1
	2 buses 4 clusters	2	16	16	∞	16	2
	2 buses 2 clusters	4	32	32	∞	16	2
`sendb` + 4/8/16-entry CRB	1 bus 4 clusters	2	16	16	[4/8/16]x4	16	1
	1 bus 2 clusters	4	32	32	[4/8/16]x4	16	1
	2 buses 4 clusters	2	16	16	[4/8/16]x4	16	2
	2 buses 2 clusters	4	32	32	[4/8/16]x4	16	2

outperforms the traditional `rcopy` scheme. In 4-cluster configurations, on an average an additional speedup of 2%, 4.9%, and 5.7% are observed over `rcopy` scheme when 4, 8 and 16-entry CRB is used with partitioned register file respectively. Speedup higher than one was observed on clustered machines for some benchmarks (124.m88ksim). This is primarily due to the non-linearity of the cluster-scheduling process and also due to the aggressive peephole optimizations done after CARScheduling; similar observations have been reported for the code generated using other algorithms [12]. For some benchmarks, especially for 126.gcc and 134.perl, the `rcopy` scheme is observed to perform better than `sendb` scheme. This is because of the combined effect of factors such as finite size of CRB, and redundant sendb OPs scheduled for some long live ranges with multiple reaching Defs. It may be possible to eliminate these `sendb` OPs (or replace them with a single local `sendb` OP) by making an additional pass on the scheduled code performing a data flow analysis. A partitioned register file with only a 4-entry CRB is therefore sufficient to achieve a performance identical to a clustered machine with an infinitely large local register space for keeping remote registers. Note that the rcopy experiments provide a conservative comparison: we have not modeled the effects of increase in register pressure, the potential possibility of spills due to the increased register pressure, and the impact of increase in code size in our `rcopy` experiments. These effects would have further lowered the performance of the `rcopy` scheme reported herein.

5 Related work

The partitioned register file structures used in prior clustered ILP processor architectures [9, 11, 4, 10, 2] are different from our CRB-based scheme as explained in section 1. Caching of processor registers in general is not a new concept and has been used in different contexts such as the Named-State Register file [16] and Register-Use Cache [17] for multi-threaded processors, the pipelined register cache [18], and Register Scoreboard and Cache [19]. Our scheme is different from these as none of them are aimed at keeping a local copy of a remote register.

Fernandes *et al.* [20] proposed a partitioned register file scheme in which individually addressable registers are replaced by queues. In contrast, our scheme

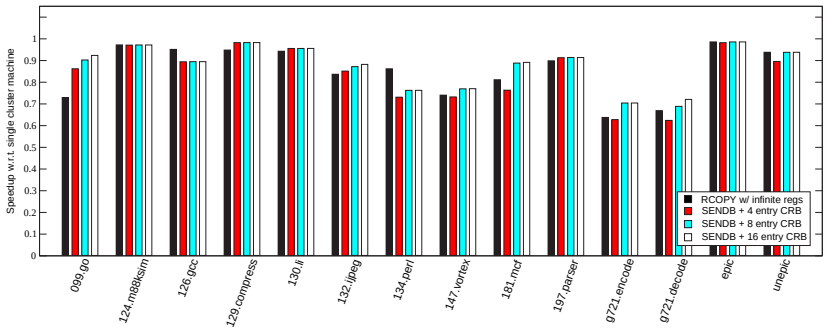

Fig. 4. Speedup for different CRB and `rcopy` configurations over single cluster for 1 bus 2 clusters configuration

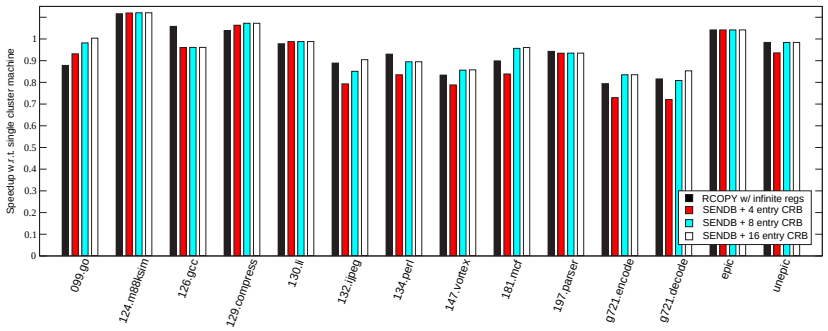

Fig. 5. Speedup for different CRB and `rcopy` configurations over single cluster for 2 buses 2 clusters configuration

permits random access to the cached remote registers in the CRB, and also provides more freedom to the cluster scheduler for scheduling the Use-nodes of remote registers in any arbitrary order limited only by true data dependencies. Llosa *et al.* proposed a dual register file scheme which consists of two replicated, yet not fully consistent register files [21]. Cruz *et al.* proposed a multiple-bank register file [22] for dynamically scheduled processors in which the architected registers are assigned at run time to multiple register banks. Zalamea *et al.* have proposed a two-level hierarchical register file [23] explicitly managed using two new spill instructions. In contrast, CRB is transparent to the register allocator; it is not assigned any architected name-space, which in turn allows it to cache any architected register. Due to space limitations, readers are referred to [8] for a more comprehensive list of related work and comparison of those with our scheme.

6 Conclusion

We presented a new partitioned register file architecture that uses a caching register buffer (CRB) structure to cache the frequently accessed remote registers without using any architected registers in the local register file. The scheme reduces register pressure without increasing the architected name space (which would have necessitated increasing the number of bits used for specifying register operands in instruction encoding). The CRB structure requires only one write

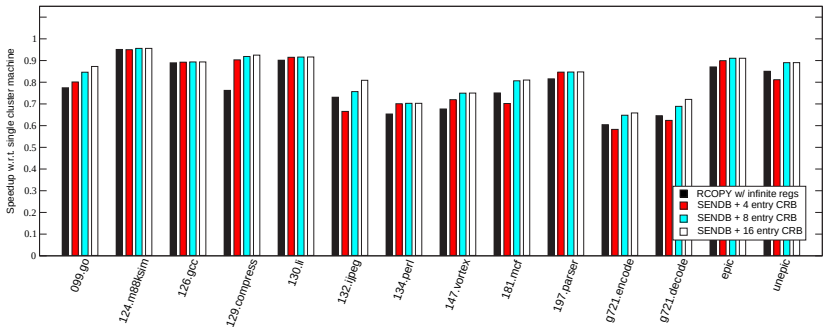

Fig. 6. Speedup for different CRB and `rcopy` configurations over single cluster for 1 bus 4 clusters configuration

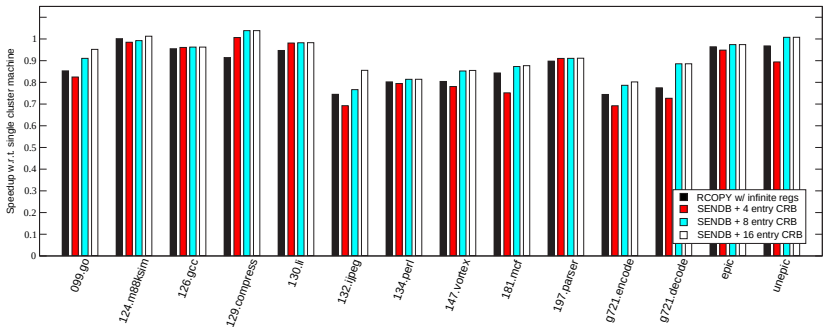

Fig. 7. Speedup for different CRB and `rcopy` configurations over single cluster for 2 buses 4 clusters configuration

port per inter-cluster communication bus, and the partitioned register file with CRB can be realized without affecting register file access time of a clustered ILP processor. The CRB is managed explicitly by the compiler using a new `sendb` operation. The `sendb` OP can update multiple CRBs concurrently via broadcasting the register value over inter-cluster communication bus/network. The `sendb` OP can thus eliminate multiple send-receive OP pairs used in prior schemes. It can also combine several inter-cluster copy OPs used by the `rcopy`-based prior partitioned register file schemes. Experimental results indicate that in a clustered ILP processor with non-trivial number of clusters, a 4-entry windowed CRB structure can provide the same performance as that of a partitioned register file with infinite non-architected register space for keeping remote registers. Because clustered ILP processors are an attractive complexity-effective alternative to contemporary monolithic ILP processors, these results are important in the design of future processors.

References

1. V. Zyuban and P. M. Kogge, "Inherently Lower-Power High-Performance Superscalar Architectures," *IEEE Trans. on Computers*, vol. 50, pp. 268–285, Mar. 2001.
2. P. Faraboschi, J. Fisher, G. Brown, G. Desoli, and F. Homewood, "Lx: A Technology Platform for Customizable VLIW Embedded Processing," in *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

3. K. Ebcioğlu, J. Fritts, S. Kosonocky, M. Gschwind, E. Altman, K. Kailas, and T. Bright, "An Eight-Issue Tree-VLIW Processor for Dynamic Binary Translation," in *Proc. of Int. Conf. on Computer Design (ICCD'98)*, pp. 488–495, 1998.
4. Texas Instruments, Inc., *TMS320C62x/C67x Technical Brief*, Apr. 1998.
5. J. Fridman and Z. Greenfield, "The TigerSHARC DSP architecture," *IEEE Micro*, vol. 20, pp. 66–76, Jan./Feb. 2000.
6. R. E. Kessler, "The Alpha 21264 microprocessor," *IEEE Micro*, vol. 19, pp. 24–36, Mar./Apr. 1999.
7. R. Canal, J. M. Parcerisa, and A. Gonzalez, "Dynamic cluster assignment mechanisms," in *Proc. of the 6th Int. Conference on High-Performance Computer Architecture (HPCA-6)*, pp. 133–142, Jan. 2000.
8. K. Kailas, *Microarchitecture and Compilation Support for Clustered ILP Processors*. PhD thesis, Dept. of ECE, University of Maryland, College Park, Mar 2001.
9. R. P. Colwell *et al.*, "A VLIW architecture for a trace scheduling compiler," *IEEE Transactions on Computers*, vol. C-37, pp. 967–979, Aug. 1988.
10. S. Keckler, W. Dally, D. Maskit, N. Carter, A. Chang, and W. Lee, "Exploiting fine-grain thread level parallelism on the MIT Multi-ALU processor," in *Proc. of the 25th Annual Int. Symposium on Computer Architecture*, pp. 306–317, 1998.
11. A. Capitanio, N. Dutt, and A. Nicolau, "Partitioned register files for VLIWs: A preliminary analysis of tradeoffs," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 292–300, Dec. 1–4, 1992.
12. K. Kailas, K. Ebcioğlu, and A. Agrawala, "CARS: A New Code Generation Framework for Clustered ILP Processors," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA-7)*, pp. 133–143, 2001.
13. K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill, "Dependence flow graphs: an algebraic approach to program dependencies," in *Proc. of the 18th annual ACM symposium on Principles of programming languages*, pp. 67–78, 1991.
14. R. Sethi, *Algorithms for minimal-length schedules*, ch. 2. Computer and job-shop scheduling theory (E. G. Coffman, ed.), John Wiley & Sons, Inc., New York., 1976.
15. M. Moudgill, "Implementing an Experimental VLIW Compiler," *IEEE Technical Committee on Computer Architecture Newsletter*, pp. 39–40, June 1997.
16. P. R. Nuth, *The Named-State Register File*. PhD thesis, MIT, AI Lab, Aug. 1993.
17. H. H. J. Hum, K. B. Theobald, and G. R. Gao, "Building multithreaded architectures with off-the-shelf microprocessors," in *Proceedings of the 8th International Symposium on Parallel Processing*, pp. 288–297, 1994.
18. E. H. Jensen, "Pipelined register cache." U.S. Patent No. 5,117,493, May 1992.
19. R. Yung and N. C. Wilhelm, "Caching processor general registers," in *International Conference on Computer Design*, pp. 307–312, 1995.
20. M. M. Fernandes, J. Llosa, and N. Topham, "Extending a VLIW Architecture Model," technical report ECS-CSG-34-97, Dept. of CS, Edinburgh University, 1997.
21. J. Llosa, M. Valero, and E. Ayguade, "Non-consistent dual register files to reduce register pressure," in *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pp. 22–31, 1995.
22. J.-L. Cruz, A. González, M. Valero, and N. P. Topham, "Multiple-banked register file architectures," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 316–325, 2000.
23. J. Zalamea, J. Llosa, E. Ayguade, and M. Valero, "Two-level hierarchical register file organization for VLIW processors," in *Proceedings of the 33rd Annual International Symposium on Microarchitecture (MICRO-33)*, 2000.