Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures

Emre Özer
North Carolina State University
Raleigh, NC 27695
`eozer@eos.ncsu.edu`

Sanjeev Banerjia
Hewlett-Packard Laboratories
Cambridge, MA 02142
`sanjeev@hpl.hp.com`

Thomas M. Conte
North Carolina State University
Raleigh, NC 27695
`conte@eos.ncsu.edu`

Abstract

Recently, there has been a trend towards clustered microarchitectures to reduce the cycle time for wide-issue microprocessors. In such processors, the register file and functional units are partitioned and grouped into clusters. Instruction scheduling for a clustered machine requires assignment and scheduling of operations to the clusters. In this paper, a new scheduling algorithm named unified-assign-and-schedule *(UAS) is proposed for clustered, statically-scheduled architectures. UAS merges the cluster assignment and instruction scheduling phases in a natural and straightforward fashion. We compared the performance of UAS with various heuristics to the well-known Bottom-up Greedy (BUG) algorithm and to an optimal cluster scheduling algorithm, measuring the schedule lengths produced by all of the schedulers. Our results show that UAS gives better performance than the BUG algorithm and is quite close to optimal.*

1 Introduction

Many high performance processors are designed with wide issue widths to exploit high levels of instruction level parallelism (ILP). Wide issue machines require a large number of functional units and a large register file. The register file should provide a sufficient number of read and write ports to the functional units. However, a large number of register file ports can increase the cycle time of the processor [1], [2]. One way to avoid this is to partition the register file and distribute it among disjoint sets of functional units, preventing any decrease in cycle time. Unfortunately, partitioning requires additional copy operations (via hardware or software) to keep the files coherent. A set of functional units and its associated register file partition is term a *cluster*. Each cluster is connected to an interconnection network to allow communication with other clusters. Such a machine organization can be termed a clustered microarchitecture, as shown in Figure 1.

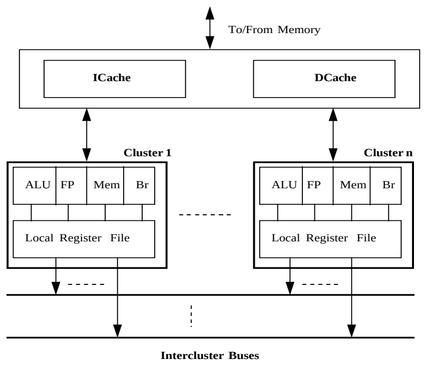


Figure 1. A sample clustered microarchitecture. Current similar microarchitectures include the Alpha 21264 and the TI TMS 320C6x. The disjoint nature of the register files complicates instruction scheduling.

Compiling for a clustered microarchitecture requires additional steps compared to compiling for a conventional microarchitecture which has a central register file. For convenience, we term compilation for a clustered microarchitecture *cluster scheduling*. Cluster scheduling's additional tasks are to coordinate and schedule the movements of data among clusters and the assignment of data and operations to specific clusters (which is termed *cluster assignment*). Cluster scheduling can be critical to the performance of a clustered machine. A cluster scheduler should use the machine resources effectively to exploit ILP and hide the disjoint nature of the register file completely. Cluster scheduling decisions are crucial especially for Very Long Instruction Word (VLIW) style machines that rely almost entirely on compiler technology [3], [4]. Clustering is also used in dynamically scheduled processors, such as the Digital Alpha 21264 [2].

This paper examines instruction scheduling for clustered processors, focusing on the cluster assignment problem for VLIW architectures. A novel technique that uses a single compiler phase – Unified Assign and Schedule (UAS) – is presented and evaluated experimentally. We also present an optimal cluster scheduler to find out how heuristic clustering techniques, such as UAS and Bottom-up Greedy (BUG) [5], approximate optimality. Finally, the performance of UAS is compared to BUG. The organization of the paper is as follows. Section 2 examines related work in cluster scheduling. Section 3 introduces the UAS algorithm and its implementation. Section 4 introduces the optimal cluster scheduler for a clustered VLIW architecture and compares it to UAS and BUG. Section 5 presents performance analysis of UAS and BUG. Finally, Section 6 concludes the paper with some observations.

2 Related work

There have been several previous projects that dealt with compile-time cluster scheduling. The Bulldog compiler was the first attempt to examine the effects of clustering. The compiler implements trace scheduling [6] and makes cluster assignments to the operations in the trace. List scheduling is then applied to the trace to construct a schedule of instructions. Bulldog uses a multi-phase approach to cluster scheduling. Separate compiler phases are employed to perform trace formation, cluster assignment, and list scheduling. The cluster assignment phase uses the Bottom-up Greedy (BUG) algorithm to assign operations and register values to clusters. It takes the data precedence graph (DPG) of a trace as its input and traverses it from the roots (exit nodes) to the leaves (entry nodes) in a bottom-up fashion. It recursively traverses the DPG and makes estimates about functional unit and operand availability for each operation. After BUG assigns the operations in a trace, the list scheduler inserts communication operations into the schedule where necessary.

BUG is an intuitive algorithm that has a complete resource model. Since it does not produce the final code schedule, it cannot anticipate what the actual resource usage pattern will be during scheduling, nor does it keep track of utilization of the intercluster buses. Therefore, the cluster assignments made by the BUG algorithm are based on rough estimates, a drawback of the algorithm. The Multiflow TRACE compiler [7] was based on technology implemented within Bulldog but with significant differences, including its implementation of BUG. The Multiflow compiler tries to place operations in a dependence chain in the same cluster and spread the chains to clusters evenly to increase parallelism. However, it cannot eliminate the oversaturation of the interconnection buses since it does not have complete knowledge of interconnect availability.

Another work in cluster scheduling is Limited Connectivity VLIW [8], [9]. The focus of the work is code partitioning for a clustered VLIW machine which does not have full connectivity between all registers and functional units. The technique was implemented within the Percolation Scheduling compiler developed at UC-Irvine [10]. It uses a multi-phase approach similar to Bulldog. Initially, code is scheduled assuming that the machine is a fully connected VLIW, i.e., all functional units can access all registers directly. Then, the code is partitioned and cross-partition copy operations are inserted. The code is then compacted locally to minimize the effect of inserted copy operations in the schedule.

Desoli's Partial Component Clustering (PCC) [11] algorithm uses a multi-phase iterative approach to perform cluster assignment. The first part of the algorithm decomposes a DAG into multiple smaller DAGs (partial components) using a depth-based DAG traversal much like BUG. The partial components are processed in decreasing order of the number of nodes to derive an initial set of cluster assignments. The initial assignment logic uses load balancing (the number of nodes per cluster) as a primary criterion and number of intercluster copies as a secondary criterion. The set of initial assignments is processed by a descent algorithm that iteratively attempts to improve the assignments. The descend phase use lightweight list scheduling-like logic to locally modify the set of initial assignments until no reductions in schedule length or the number of copies can be achieved. The lightweight scheduling logic accounts for register usage and the effects of copy instructions. After the descend phase completes, the set of assignments is actually scheduled by a list scheduling/register allocation phase. PCC differs from UAS in several major facets: 1) separate phases are used for assignment and scheduling, 2) global pre-processing DAG analysis is employed, and 3) an iterative algorithm is used.

The RAW project uses static instruction assignment and scheduling for its highly parallel distributed architecture [12]. A RAW machine is very similar to a VLIW with the major point of departure being that a RAW machine can exploit multiple flows of control in parallel. The RAW compiler, RAWCC, uses multiple phases for cluster assignment. Leveraging work from multiprocessor scheduling, it uses the Dominant Sequent Clustering (DSC) algorithm [13] to delineate a DAG into multiple instruction streams called clusters. As described in a recent work, their implementation of DSC behaves very similarly to BUG: a topological traversal of the dependence graph attempts to place each instruction near one of its data-dependent predecessors, assuming zero communication costs. Next comes a merging phase that combines the multiple instruction streams (clusters) so that the resulting number of clusters does not exceed the number of execution elements in the machine. Merging uses the amount of inter-cluster communication as a primary criterion and load balancing as a secondary criterion. The final binding of clusters to machine elements is performed by first assigning clusters to elements randomly and then employing a swapping phase to attempt to reduce the amount of communication.

There has been related work in compiler support for clustering in the context of superscalar processors. As these works are not focused on statically-scheduled architectures, we discuss them briefly. Farkas proposed the MultiCluster architecture as a design that uses clusters to reduce cycle time slowdowns [14]. Compile-time analysis is used to assign register operands to clusters with a primary goal of achieving balance in the number of instructions assigned per cluster at any point in time. Hardware support is used to detect when data movements are required and then coordinate them. Sastry et al. [15] explored instruction assignment by viewing a superscalar processor as a microarchitecture composed of two clusters, one of integer units and the other of floating point (FP) units. Their objective was to use idle FP resources to execute integer instructions by assigning some instructions that normally execute on the integer cluster to the FP cluster. Two instruction partitioning schemes were introduced, both of which performed instruction assignment as pre-pass. A notable trait of the more sophisticated scheme is that it permits the insertion of explicit copy operations and code duplication. A cost/benefit analysis is applied to every instruction that could be assigned to the FP cluster by examining its predecessors to compute the number of extra instructions created by either copying or duplication. both the basic and advanced schemes. The cost/benefit scheme used could be applied or modified for use with VLIW architectures as well.

There are also novel cluster-based architectures that are less heavily focused on compile-time cluster assignment, such as Multiscalar [16] and the M-Machine [17]. Since our focuses on compile-time assignment, these works are not discussed further. Future work is planned for adapting UAS to dynamically scheduled microarchitectures.

3 Unified Assign and Schedule

Schedule-time resource availability is not checked in BUG or Limited Connectivity VLIW since cluster assignment is performed in a different compiler phase than scheduling. This leads to clustering decisions that either constrain scheduling (in case of BUG) or are needlessly constrained by prior scheduling choices (in case of Limited Connectivity VLIW), as shown in Figure 2.

An algorithm that performs assignment and scheduling within one, unified phase could avoid these drawbacks. We term an algorithm that performs cluster assignment and scheduling as *Unified-Assign-and-Schedule* or UAS [18]. UAS essentially integrates cluster assignment into a list scheduler. List scheduling is the most common technique for instruction scheduling in production compilers. A list scheduler schedules as many operations as possible for the current cycle before processing the next cycle. A list of operations and a DPG of the list are passed to the scheduler. The list

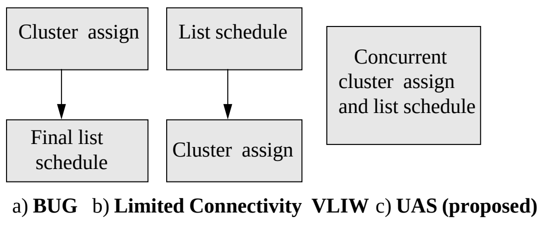


Figure 2. The three approaches to cluster scheduling: (a) BUG, (b) Limited Connectivity VLIW, and (c) the proposed UAS approach.

is ordered based on some known priority function. The algorithm consists of two main loops. The outer loop ensures that all operations are scheduled. The inner loop schedules as many operations as possible into the current cycle. Once a cycle is scheduled, it is never revisited. Cluster assignment is integrated into the inner loop of the list scheduler. Intercluster buses are considered to be machine resources and checked within the list scheduler when the other resources are checked.

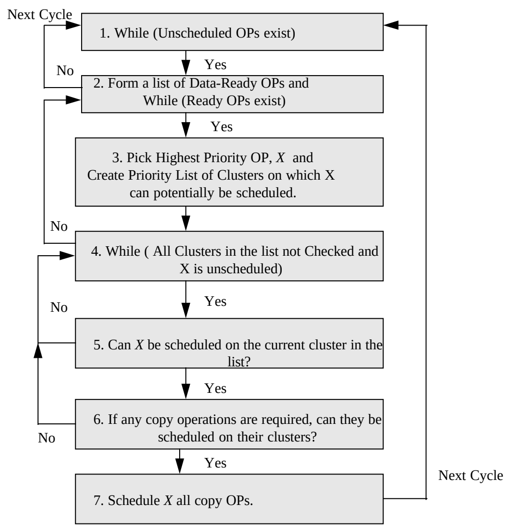

Next Cycle
1. While (Unscheduled OPs exist)
Yes
No
2. Form a list of Data-Ready OPs and While (Ready OPs exist)
Yes
3. Pick Highest Priority OP, X and Create Priority List of Clusters on which X can potentially be scheduled.
No
4. While (All Clusters in the list not Checked and X is unscheduled)
Yes
No
5. Can X be scheduled on the current cluster in the list?
Yes
6. If any copy operations are required, can they be scheduled on their clusters?
No
Yes
7. Schedule X all copy OPs.
Next Cycle

Figure 3. A framework for implementing unified assign and schedule(UAS).

A detailed explanation of UAS algorithm is given in Figure 3. First, a list of data-ready operations is formed for each cycle (Step 1). Formation of such a list is slightly different from a list scheduler in that inter-cluster copy latencies are also taken into consideration. For instance, if there are at least two flow-dependent predecessors assigned to different clusters, then at least one inter-cluster copy is required to place all of the source operands of the operation under consideration onto a single cluster. So the effective weight of a flow-dependency edge is the predecessor operation latency plus the inter-cluster communication latency. In Step 3, a prioritized list of clusters is formed, and the highest priority operation is picked as a candidate for scheduling. Each cluster in the prioritized list of clusters is examined in Step 5 to determine whether the current operation can be scheduled on it. When an available cluster is found, a check is made to see if any copy operations are required. If the copies can be scheduled on their respective clusters, *i.e.*, there are enough available intercluster buses in the current cycle) in Step 6, the current operation and associated copies are scheduled (Step 7). If not, the next cluster is examined for the current operation. This continues until the cluster list is exhausted or the operation is successfully scheduled.

The priority function for ordering the list of cluster ids can have a strong effect on the schedule. There are many different priority functions that can be used. Five are investigated in this paper:

- **None**: The cluster list is not ordered.
- **Random Ordering**: The cluster list is ordered randomly.
- **Magnitude-weighted Predecessor (MWP)**: The number of flow-dependent predecessors assigned to each cluster for the operation under consideration is computed. Based on the counts, an operation can be placed into a cluster where the majority of its input operands reside.
- **Completion-weighted Predecessor(CWP)**: Each flow-dependent predecessor has a ready time associated with it, which is the cycle when it produces its result. Each cluster id can be weighted by the latest ready time value for the operation under consideration. The cluster list is sorted in descending order of the ready times. This gives priority to the clusters that will be producing source operands for the operation late in the schedule.
- **Critical-Path in Single Cluster using CWP Heuristics (CPSC)**: The CWP heuristic is used but operations on the critical path are forced to be assigned to the same cluster.

Certainly additional priority functions can be derived. The five listed are are simple to compute in terms of space and time. The UAS algorithm uses these priority functions for cluster ordering in the experimental evaluation.

UAS has complete information about the machine resources' status so that assignments can be performed and copies can be scheduled effectively. Although BUG has complete knowledge of all machine resources, interconnect availability is not and cannot be checked, which can lead to oversaturation of the buses during list scheduling. Hence, intuitively UAS should outperform BUG. This is substantiated empirically in Section 5.

4 Comparing UAS and BUG to optimal cluster scheduling

We implemented simultaneous optimal partitioning, assignment and scheduling of a given DPG for clustered VLIW architectures to seek the best cluster assignment and schedule. Optimal partitioning and scheduling of a DPG is an NP complete problem [19]. Thus, optimal cluster assignment (partitioning) and scheduling is also NP complete. It is impractical to implement optimal cluster scheduling within compilers because of the exponential time complexity. The intent here behind seeking optimality is to find an upper bound for a heuristic-based scheme. Our objective is to find a best possible cluster assignment of all operations in a DPG that minimizes the schedule length.

The optimal cluster scheduler is designed as a resource-constrained scheduler that attempts to find the shortest schedule possible. Given a DPG of operations, the optimal cluster scheduler finds optimal cluster assignments and operation scheduling based on machine resource and data dependence constraints, such as the number of functional units per cluster, the number of intercluster buses, and the partial order between operations. We formulate resource constraints and the objective function (schedule length) as linear equations. The equations are solved by 0-1 integer programming (IP) using a branch and bound technique. Simultaneous assignment and scheduling has also been studied in the VLSI synthesis field [20], [21]. The framework used for this study is shown in Figure 4.

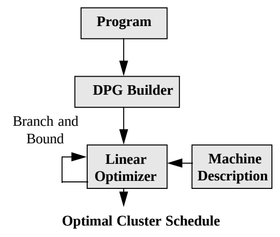


Figure 4. General framework for optimal cluster scheduling.

The compiler takes a block of operations, forms a DPG, annotates extra information such as the early and late times of operations, and then dispatches the DPG to the Linear Optimizer. The Linear Optimizer reads the machine description and generates linear equations of all resource constraints and the objective function, and then solves the equations. Linear equations are solved by the LINDO Linear Optimizer program [22]. Finally, optimal cluster schedule is compared with the schedules generated by UAS and BUG.

Resource constraints for a clustered VLIW machine consist of the operations, functional units, cluster assignment and partial order of operations, and intercluster bus constraints. The operation constraint ensures that each operation is given a schedule time and is assigned to a functional unit within a cluster. Functional unit constraints ensure that no more than the total number of a functional unit type is allocated for operations for a certain cycle. For example, if there is only one ALU unit per cluster in a two cluster machine, at most two integer operations at a time can be scheduled on those ALUs. Cluster assignment constraints assign operations to clusters and schedules them by honoring the dependence order and inter-cluster latency. If two operations with a flow dependence edge between them are assigned to the same cluster, no copy is needed. However, if they are assigned to different clusters, inter-cluster communication is necessary Bus constraints guarantee that no more than total number of buses are used for intercluster communication during a certain cycle. We assume that all functional units are fully pipelined and that any operation can be assigned to any cluster. The objective function attempts to minimize the schedule length for the DPG while honoring these constraints. Details about the linear equations can be found in companion technical report [23].

We selected programs from the SPECint95 [24] and MediaBench [25] suites as listed in Table 1. Programs from SPECint95 represents general-purpose integer programs. Programs from MediaBench focus on media and signal processing. Reference inputs were used for all runs.

Table 1. Benchmarks programs used for evaluation.

SPECint95	MediaBench
132.ijpeg	g721decode
147.vortex	g721encode
129.compress	rawcaudio
130.li	rawdaudio
124.m88ksim	
099.go	
134.perl	

The machine models used in the experiments are 2 cluster 4 issue VLIWs with one bus and two buses. Each cluster has four functional units (ALU, FP, BR and LD/ST units). All clusters are assumed to have an identical configuration. The inter-cluster communication latency is one cycle. The functional unit latencies are shown in Table 2.

The UAS algorithm with heuristics CPSC, CWP, MWP, Random, and NONE and the BUG algorithm are compared to the optimal cluster scheduler (OPT). A large number of possible cluster schedules are normally evaluated to find the optimal schedule. It takes a considerable time to find the optimal solution for full benchmark runs. Instead, we selected the most frequently executed basic block with a reasonable number of operations from each benchmark and measured its static schedule length.

Table 2. Instruction types and latencies

Instruction Type	Latency
Integer/Branch, Store, FP Add	1
Load	2
FP Multiply	3
FP Divide	9

Results are presented in Figure 5. The optimal cluster schedule is our theoretical upper bound, so speedup values are expected to be less than 1. The results show that UAS generates more compact schedules than BUG for all benchmarks. Also, in Figure 6 we present code increases for OPT, UAS and BUG. Code increase is measured as the percentage increase in code size due to the number of copy operations for the scheduled set of basic blocks. One observation is that the optimal cluster scheduler attempts to distribute operations across clusters to utilize resources and extracts more parallelism. Therefore the number of inter-cluster bus communications is high. UAS also spreads operations among clusters and has a good cluster utilization without compromising much performance, except for the *Random* heuristic. Another observation is that the BUG algorithm does not utilize clusters effectively. It uses only one cluster for all operations in most of the selected blocks. Thus, it cannot extract as much ILP parallelism as UAS.

5 Full benchmark comparison of UAS and BUG

This section presents the results of experiments to evaluate the performance of UAS and its comparison to BUG algorithm for entire benchmark runs. Instead of basic blocks, treegions [26] are used for full benchmark runs. The scheduling algorithms were implemented within the framework of the LEGO compiler [26], [27]. A static estimate based on schedule length is used to measure the performance of code scheduled by LEGO. We measured speedups of UAS and BUG on a clustered machine with respect to code scheduled for a base model that is 1-cluster machine with the same number of functional units. Perfect instruction and data caches and perfect branch prediction are assumed throughout the experiments. We used three different machine models: 2 cluster 4 issue (ALU, FP, BR, LD/ST), 2 cluster 2 issue (2 UNIVERSAL Functional Units) and 3 cluster 2 issue (2 UNIVERSAL Functional Units). All three models have one inter-cluster bus (little performance increase is observed with two buses).

The results are shown in Figure 7. The figure shows speedups of UAS and BUG with respect to the base machine model. The base machine model is one cluster with the same type and number functional units for each machine model. As seen from the graphs, UAS outperforms BUG across the entire set of benchmarks. All UAS heuristics have better performance than BUG except for *Random*, since a random ordering of clusters is more likely to allow an operation's flow-dependent operations to be assigned to different clusters. This could increase the number of copies and the schedule length unnecessarily. On average, UAS with the CWP heuristic yields the best performance results, even better than CPSC heuristic. This is because blindly assigning critical-path operations to the same cluster may cause bad cluster assignments for operations in the other paths.

6 Conclusion

In this paper, we have introduced unified-assign-and-schedule (UAS), a new approach to scheduling for clustered microarchitectures. UAS is a single- pass scheduler that integrates cluster assignment into a list scheduling phase. Our evaluations show that the UAS technique creates efficient schedules by using a full and complete knowledge of resources and interconnection availability among the clusters. Hence, the generated schedule is compact, efficient and also relatively close to optimal. As microarchitecture designers continue to design processors with disjoint register files the effectiveness of UAS over other approaches will become critical.

References

[1] R. P. Colwell, R. P. Nix, J. J. O'Donnel, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," in *Proc. Second Int'l. Conf. Architectural Support for Programming Languages and Operating Systems.*, (Palo Alto, CA), pp. 180–192, Oct. 1987.

[2] L. Gwennap, "Digital 21264 sets new standard," *Microprocessor Report*, vol. 10, Oct. 1996.

[3] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau, "Parallel processing: A smart compiler and a dumb machine," in *Proc. 1984 SIGPLAN Symp. Compiler Construction*, pp. 37–47, June 1984.

[4] Texas Instruments, Inc., *TMS320C62xx CPU and Instruction Set: Reference Guide.* Texas Instrumentc, Inc., July 1997. Manufacturing part # D426008-9761, revision A.

[5] J. R. Ellis, *Bulldog: A compiler for VLIW architectures.* Cambridge, MA: The MIT Press, 1986.

[6] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. C-30, no. 7, pp. 478–490, July 1981.

[7] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donell, and J. C. Ruttenberg, "The Multiflow Trace scheduling compiler," *J. Supercomputing*, vol. 7, pp. 51–142, Jan. 1993.

[8] A. Capitanio, N. Dutt, and A. Nicolau, "Partitioned register files for VLIWs: a preliminary analysis," in *Proc. 25th Ann. Int'l Symp. Microarchitecture*, (Portland, OR), pp. 292–300, Dec. 1992.

[9] A. Capitanio, N. Dutt, and A. Nicolau, "Design considerations for Limited Connectivity VLIW architectures," Tech. Rep. TR59-92, Department of Information and Computer Science, University of California at Irvine, Irvine, CA 92717, 1992.

[10] R. Potasman, *Percolation-Based Compiling for Evaluation of Parallelism and Hardware Design Trade-Offs.* PhD thesis, Department of Computer and Information Sciences, The University of California at Irvine, Irvine, CA, 1991.

[11] G. Desoli, "Instruction assigment for clustered VLIW DSP compilers: a new approach," Tech. Rep. HPL-98-13, Hewlett-Packard Laboratories, 1998.

[12] W. Lee, R. Barva, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe, "Space-time scheduling of instruction-level parallelism on a RAW machine," in *Proc. Eighth Int'l. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, (San Jose, CA), Oct. 1998.

[13] T. Yang and A. Gerasoulis, "Dsc: Scheduling parallel tasks on an unbounded number of processors," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 9, pp. 951–967, 1994.

[14] K. I. Farkas, *Memory-system design considerations for dynamically-scheduled microprocessors.* PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ontario, Jan. 1997.

[15] S. S. Sastry, S. Palacharla, and J. E. Smith, "Exploiting idle floating point resources for integer execution," in *Proc. ACM SIGPLAN 1998 Conf. Programming Language Design and Implementation*, (Montreal), pp. 118–129, June 1998.

[16] G. S. Sohi, S. E. Breach, and T.N.Vijaykumar, "Multiscalar Processors," in *Proc. 22nd Ann. Int'l Symp. Computer Architecture*, (Santa Margherita Ligure, Italy), pp. 414–425, May 1995.

[17] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee, "The M-Machine multicomputer," in *Proc. 28th Ann. Int'l Symp. Microarchitecture*, (Ann Arbor, MI), pp. 146–156, Dec. 1995.

[18] S. Banerjia, *Instruction scheduling and fetch mechanisms for clustered VLIW processors.* PhD thesis, Dept. Electrical and Computer Engineering, North Carolina State University, 1998.

[19] M. R. Garey and D. S. Johnson, *Computers and intractability : a guide to the theory of NP-completeness.* San Francisco: W. H. Freeman, 1979.

[20] J. L. C. Hwang and Y. Hsu, "A formal approach to the scheduling problem in high level synthesis," *IEEE Trans. Comput.*, vol. 10, pp. 463–475, Apr. 1991.

[21] C. Gebotys, "An optimization approach to the synthesis of multichip architectures," *IEEE Trans. Comput.*, vol. 2, pp. 11–20, Mar. 1994.

[22] `http://www.lindo.com/`.

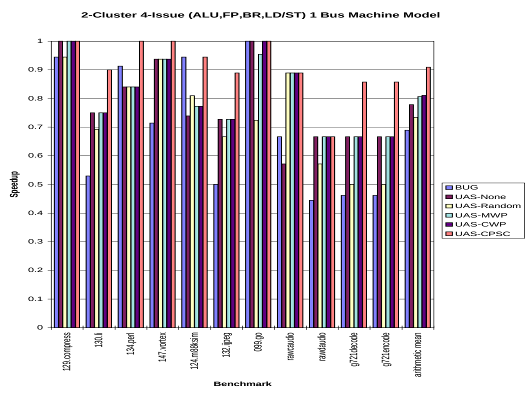
[23] E. Ozer and T. M. Conte, "Optimal cluster scheduling for a VLIW machine," tech. rep., TINKER Laboratory, Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC, June 1998.

[24] K. Dixit and J. Reilly, "SPEC95 questions and answers," *SPEC newsletter*, Sept. 1995. SPEC, Fairfax, VA.

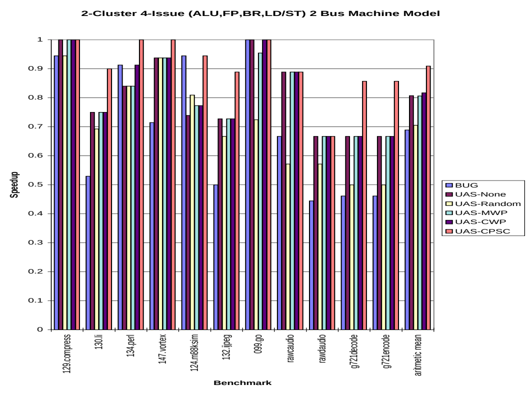
[25] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: A tool for evaluating multimedia and communications systems," in *Proc. 30th Ann. Int'l Symp. Microarchitecture*, (Research Triangle Park, NC), pp. 330–335, Dec. 1997.

[26] W. A. Havanki, "Treegion scheduling for VLIW processors," Master's thesis, Dept. Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695-7911, July 1997.

[27] W. A. Havanki, S. Banerjia, and T. M. Conte, "Treegion scheduling for wide-issue processors," in *Proc. 4th Int'l Symp. High Performance Computer Architecture*, (Las Vegas, NV), pp. 266–276, Feb. 1998.

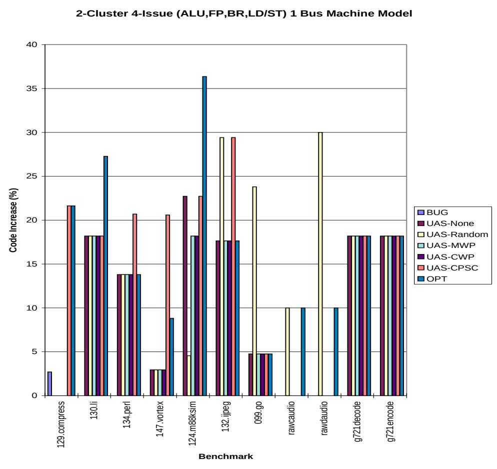


a) 2 clusters with 1 bus

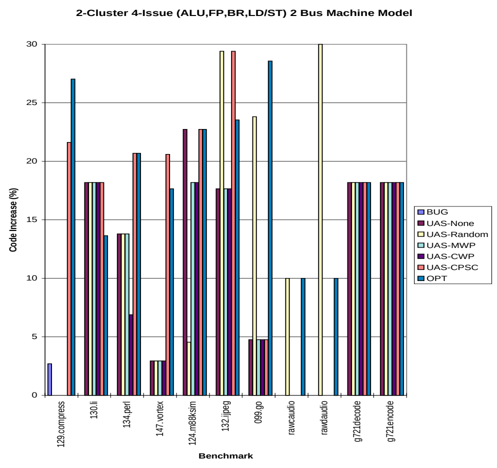


b) 2 clusters with 2 buses

Figure 5. UAS with different heuristics and BUG performance comparison to optimal cluster scheduler for a 8-wide 2 cluster VLIW machine with 1 and 2 bus models.

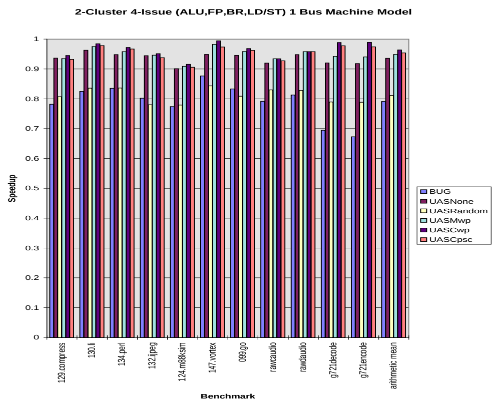


a) 2 clusters with 1 bus

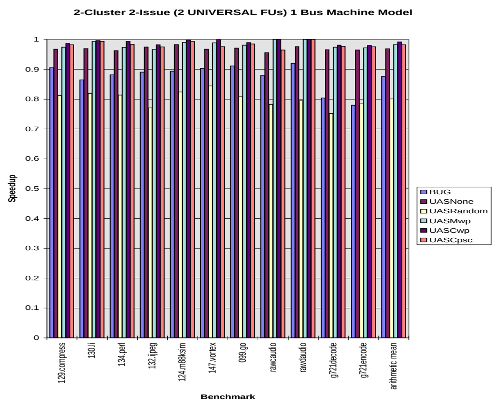


b) 2 clusters with 2 buses

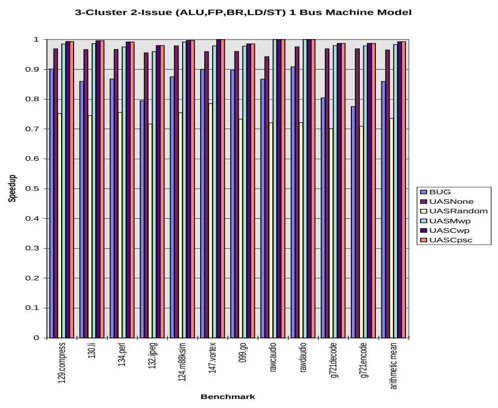
Figure 6. Comparison of code increase in UAS with its heuristics,BUG and optimal cluster scheduler(OPT).



a) 2-cluster 4-issue (ALU,FP,BR,LD/ST) with 1 bus



b) 2 cluster 2-issue (UNIVERSAL FUs) with 1 bus



c) 3 cluster 2-issue (UNIVERSAL FUs) with 1 bus

Figure 7. Speedup graphs of UAS with different heuristics and BUG for 2-cluster 4-issue (ALU,FP,BR,LD/ST),2 cluster 2-issue (2 UNIVERSAL FUs),3 cluster 2-issue (2 UNIVERSAL FUs) with 1 Bus Machine Models.