Integrated Temporal and Spatial Scheduling for Extended Operand Clustered VLIW Processors

Rahul Nagpal and Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science
Bangalore, India
{rahul,srikant}@csa.iisc.ernet.in

Abstract

Centralized register file architectures scale poorly in terms of clock rate, chip area, and power consumption and are thus not suitable for consumer electronic devices. The consequence is the emergence of architectures having many interconnected clusters each with a separate register file and a few functional units. Among the many inter-cluster communication models proposed, the extended operand model extends some of operand fields of instruction with a cluster specifier and allows an instruction to read some of the operands from other clusters without any extra cost.

Scheduling for clustered processors involves spatial concerns (where to schedule) as well as temporal concerns (when to schedule). A scheduler is responsible for resolving the conflicting requirements of aggressively exploiting the parallelism offered by hardware and limiting the communication among clusters to available slots. This paper proposes an integrated spatial and temporal scheduling algorithm for extended operand clustered VLIW processors and evaluates its effectiveness in improving the run time performance of the code without code size penalty.

1. Introduction

Proliferation of embedded systems has opened up many new research issues. Design challenges posed by embedded processors are ostensibly different from those offered by general purpose system. Apart from very high performance they also demand low power consumption, low cost, and less chip area to be practical. ILP architectures have been developed to meet the high performance need. Two major ILP design philosophies are superscalar architecture and VLIW architecture.

Superscalar processors have dedicated hardware responsible for scheduling instructions at run time to improve the performance. They also employ a branch predictor to avoid pipeline stalls that occur in the event of control transfer. However the complicated hardware needed to carry out dynamic scheduling

leads to complicated design and high power consumption. The memory requirement for branch prediction also contributes to high power consumption, chip area, and cost of these architectures. These problems of superscalar processors make them less suitable for embedded systems.

Another design philosophy is VLIW architecture where the compiler is responsible for scheduling. This simplifies the hardware but in order to exploit the ILP in media applications and to satisfy the high performance requirements of these applications, we need more functional units that can operate in parallel. This in turn requires more no of read and write ports and hence increased chip area, cycle time, and power consumption, since for N arithmetic units area of register file grows as N^3, delay as $N^{3/2}$, and power dissipation as N^3 [20].

Clustering has been proposed to overcome these difficulties with centralized VLIW architecture and to make them suitable for use in embedded systems [7]. A clustered VLIW architecture has more than one register file and connects only a subset of functional units to a register file. Groups of small computation clusters are interconnected by some interconnection topology. Texas Instrument's VelociTI [11], HP/ST's Lx [8], Analog's TigerSHARC [9], and BOPS' ManArray [19] are examples of the recent commercial micro-architectures developed based on clustered ILP philosophy. IBM's eLite [5] is a research proposal for a novel clustered architecture. Clustering avoids area and power consumption problems of centralized register file architectures while retaining high clock speed. High clock speed can be leveraged to get better performance but this demands good partitioning of computation among clusters because communication among clusters is limited and slow due to technological constraints [16] [10].

A compiler for these architecture is responsible for carrying out the complex job of mapping ILP available in an application to the parallelism offered by the underlying hardware. Specifically the scheduler is responsible for mapping the operations to resources in different clusters. The problem of scheduling becomes harder in the context of clustered architectures because a scheduler has to decide not only when to schedule (temporal concern) but also where to schedule (spatial concern). These decisions are often very complicated because scheduling an instruction at a spatial location affects temporal and spatial scheduling decisions of instructions dependent on this instruction. In order to carry out effective cluster scheduling, a scheduler is required to accurately gather and effectively utilize information regarding availability of operands in a cycle, availability of resources and cross-path in current and earlier cycles, as well as resource and cross-path requirements that may arise in the future. Essentially, effective scheduling of such an architecture demands resolving the conflicting goals of exploiting hardware parallelism as well as minimizing communication among clusters by understanding the proper trade-off between parallelism and locality and utilizing all the free-of-cost facilities provided by hardware to the maximum possible extent.

Among many inter-cluster communication (ICC) models [22] that have been proposed, the extended operand model attaches a cluster id field with some of the operands and this allows an instruction to read some of operands from register files of other clusters *without any extra delay*. Extended operand ICC model reduces the register pressure as the transferred value is not stored in the local register file but is consumed immediately. However reuse of transferred value is not possible. Texas Instruments VelociTI architecture [21] uses this model of ICC. VelociTI extends some of the operands with cluster id field and restricts the inter-cluster bandwidth to two inter-cluster reads per cycle.

Two main approaches to scheduling clustered processors are phase-decoupled scheduling [3] [6] [13] and integrated scheduling [17] [15] [12]. Due to the difficulty and complexity of engineering a combined solution there have been many proposals for phase-decoupled scheduling. The phase-decoupled approach of scheduling performs spatial scheduling of instructions (partitioning into clusters) followed by temporal scheduling while adhering to earlier spatial decisions. This approach of scheduling has these shortcomings in general.

- A spatial scheduler has only an approximate knowledge (if any) of the usage of cross-paths, functional units, and load on clusters. This inexact knowledge often leads to spatial decisions which may unnecessarily constrain a temporal scheduler and lead to suboptimal schedule.
- Some of the schemes are geared towards minimizing ICC. Though this may be the right approach while scheduling on a clustered VLIW processor with only explicit move instructions for ICC, it may not produce good schedule in the case of processors with extended operand ICC model as these schemes may reduce ILP to achieve this goal.

An integrated approach towards scheduling tries to combat the phase-ordering problem by combining spatial and temporal scheduling decisions in a single phase. The integrated approach considers instructions ready to be scheduled in a cycle and available clusters in some priority order and assigns an instruction to a cluster to reduce the requisite communication or to execute an instruction at the earliest. The earlier proposal for integrated scheduling are specific about a particular ICC model and mostly target architectures that support ICC using an explicit MV operation. However an extended operand clustered data path provides the opportunity to snoop (read) the operand from another cluster apart from explicit MV operation for ICC. To make the matter further complicated the operand snooping capabilities of functional units are often limited because only limited snooping capability can be provided to attain better clock speed over an unclustered VLIW. For example, Only some of the functional units can read their operands from other clusters and not all operands of an instruction can be snooped by all functional units. Due to the high architectural cost of providing a large no of high bandwidth cross paths, any practical manifestation of clustered architectures is likely to have fewer low bandwidth cross paths. Thus the number of operands

that can be snooped among clusters in a cycle is often limited. These restrictions add to the severity of efficiently scheduling extended operand clustered processors.

A scheduler can leverage the benefits of this model by scheduling the operations among clusters in such a way that the free ICC facility can be utilized to the maximum extent. Minimizing the need for any explicit inter-cluster move operation reduces the code size apart from reducing the register pressure and makes available more free slots for other instructions. An integrated technique utilizing exact knowledge of functional unit usage and cross path usage in current and earlier cycles can outperform any pre-partitioning algorithm that strives to balance the load among the clusters by approximate calculation of load and usage of resources. The variation in the operational and communicative capabilities of resources and tight integration of communication and resource constraints makes it difficult to determine a schedule length in advance and perform effective load balancing based on that for the machine model under consideration. An integrated approach geared towards minimizing inter-cluster communication without considering future communication that may arise due to a binding or greedily executing an instruction at the earliest may also lead to sub-optimal schedule. The variation in operational and communicative capabilities of resources further requires taking into account the resource and communication requirements of other instructions ready to be scheduled in the current cycle while binding an instruction in order to carry out an effective binding. Moreover, the tight integration of communication and resource constraints renders communication cost a function of particular resource an instruction is bound to, rather than cluster and thus effective functional unit binding which has not been considered altogether in the earlier proposals becomes an important concern.

In this paper we propose a pragmatic scheme for integrated spatial and temporal spatial scheduling of clustered processors. The Scheme utilizes the exact knowledge about available communication slots, functional units and load on different clusters as well as resource and communication requirements of other instructions ready to be scheduled in a cycle known only at schedule time to performs effective functional unit binding. We observe approximately 24% performance improvement on the average over a state-of-art phase-decoupled approach to scheduling proposed by Lapinskii et al. [13] and about 11% performance improvement over an earlier integrated scheme proposed by Ozer et al. [17] without code size penalty. It utilizes the free-of-cost communication facility provided by extended operand clustered processors to reduce code size and register pressure. The proposed algorithm is generic in terms of number of clusters and numbers and types of functional units in each cluster and can easily accommodate variations in operational and communicative capabilities of resources. We have implemented these algorithms for the Texas instruments VelociTI architecture using SUIF/MACHSUIF compiler framework. The work is underway to adapt a public domain simulator for a variety of clustered VLIW configurations and

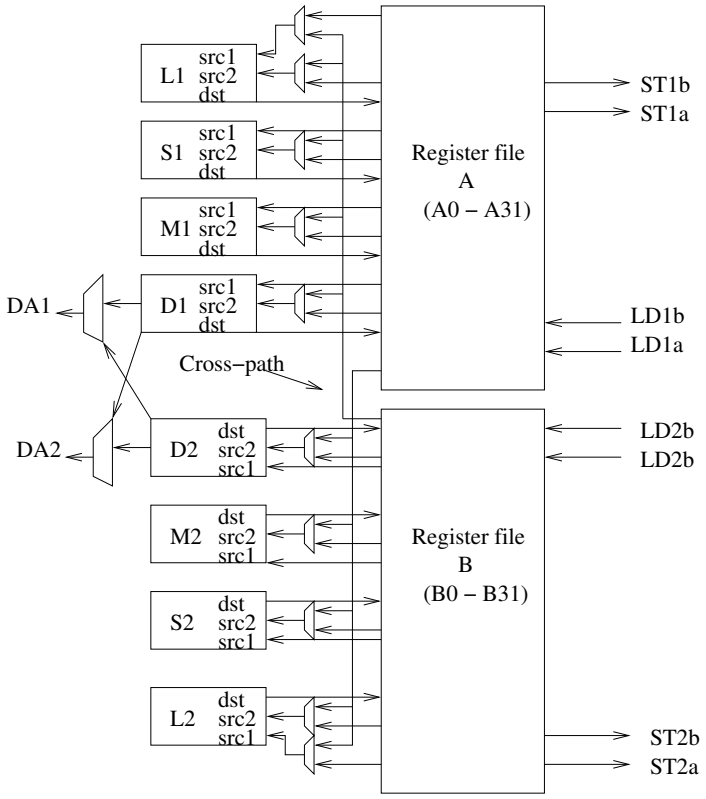

Fig. 1. TMS320C64X CPU Data Path

performance measurement. We are also in the process of adapting our algorithm for software pipelining of inner loops.

The rest of the paper is organized as follows. Section II describes the problem. Section III contains our scheduling algorithm and the implementation aspects. Section IV presents performance statistics and a detailed evaluation based on experimental data. In section V we mention related work in the area. We conclude in section VI with a mention of future work.

2. Problem Description

2.1. Machine Model

We consider a generic clustered machine model based on recently proposed clustered architectures. In our machine model a clusters can be homogeneous having identical functional units and register file (like in VelociTI architecture [21]) or heterogeneous with different cluster having a different mix of functional units and register file (like in HP Lx architecture [8]). The functional units can vary in terms of the their operational and communicative capabilities. An operation can be performed on more than one resource and a resource can perform more than one kind of operation. Some resources may also have some specialized task delegated to them. The inter-cluster communication model can vary. The architecture may even have a hybrid communication model where some of the functional units

can communicate by snooping (reading) operands from the register file of a different cluster *without any extra delay* (as in VelociTI architecture [21]), while communication among some of the clusters is possible only through an explicit MV operation (as in HP Lx architecture [8]). Snooping capabilities of functional units can be varied in terms of operands a particular functional unit can snoop as well as particular clusters with which a function unit can communicate using the snooping facility. Our machine model also incorporates architecture-specific constraints typically found in clustered architectures. For example, some operations can be performed only on some specialized resources due to performance or correctness concerns. This machine model enables us to design a generic and pragmatic framework which can accommodate architecture-specific constraints and can be easily adapted to a variety of clustered architectures differing in datapath configurations and/or communication models. Next we briefly explain the VelociTI architecture [21], having some of the features of our clustered machine model, which we have used for practical evaluation of the framework.

VelociTI is a load-store RISC-style VLIW architecture [21]. The architecture follows a homogeneous clustering philosophy and has two clusters (named A and B). Figure 1 shows the CPU data paths of TMS320C64X processor, one member of the VelociTI architecture family. Each cluster has a 32x32 register file and 4 functional units (named L, S, M and D). In addition each data path has a cross path to read operands from the other file and the address from one data path can be used to load and store values in the other data path. The ICC is restricted to two inter-cluster move one in each direction per cycle. Functional units can snoop their operands from other cluster without any extra cost. However there are some restrictions. L unit can read either of its operands from the other cluster while S, M and D can read only their second source operand from the other cluster. Most common operations are possible on four to eight units and some units have specialized operation. Explicit move operation between two clusters can be performed by blocking one of L, S or D unit for one cycle. Six registers (A0,A1,A2,B0,B1,B2) can be used for conditional execution of instructions and all instructions can be executed conditionally. However, some instructions can be conditionally executed by only specific units. The functional capabilities of each unit is as follows:

L unit	Integer addition, logical operations, and bit counting
S unit	Integer addition, logical operation, and branch control
M unit	Integer multiplication
D unit	Integer addition and load-store operation

The pipeline is divided into three phases. Fetch phase covers four pipeline stages, Decode phase covers two pipeline stages and Execute phase covers 5 pipeline stages. All functional units have a single cycle latency and can initiate a new instruction every cycle. Most of the C64x instructions are of unit latency.

16x16 bit multiply has a 2 cycles latency while load has a latency of 5 cycles and branch instruction has latency of 6 cycles. Load and store follow the same pipeline flow and thus result stored in one execute packet can be read by the next execute packet and this avoids read after write conflicts typically found in DSP processors.

2.2. The Scheduling problem

We are given a set of operation types, a set of resource types, and a relation between these two sets. This relation may not be strictly a mapping in general (and in particular in the context of VelociTI architecture), since a resource can perform more than one kind of operation and an operation can be performed on more than one kind of resource. There can be more than one instance of each type of resource. Resource instances can be partitioned into sets each one representing a cluster of resources.

Given a data flow graph, which is a partially ordered set (poset) of operations the problem is to assign each operation a time slot, a cluster and a functional unit in a chosen cluster such that the total number of time slots needed to perform all the operations in poset are minimized while the partial order of operations is honored and neither any resource nor the ICC facility is over committed.

Formally we are given

1) A set of operation types *OT*, a set of resource types *RT* and a set of clusters *C*
2) A relation between two sets given by *OR*: *OT*$\rightarrow$ *RT* such that $OR(o_i)$ for an operation o_i is a set of resource types $RT_i \subset RT$ that can perform these operations, and
3) A set *L* of triples *(t,c,r)* where *t* is a time slot, $c \in C$ and $r \in R$

A poset of operations can be represented as a directed acyclic graph G(V,E) where V is a set of operations in the poset and E represent the set of edges. Edges are labeled with a number representing the time delay needed between the head operation and the tail operation. We use the following notation:

O	Set of operations
o	An individual operations
R	Set of resources
r	An individual resource
N(RT)	Number of instance of resource type *RT*
N(c,RT)	Number of instance of resource type *RT* in cluster *c*
BW(ci,cj)	Communication bandwidth between cluster *i* and cluster *j*
NT(t,ci,cj)	Number of transfer between cluster *i* and cluster *j* in a time step t
RT(r)	Type of resource *r*
OT(o)	Type of operation *o*
C(r)	Cluster of resource *r*
l	An individual triple of *L*
ES(o_i)	The subset of edges having o_i as successor

Scheduling in this context is the problem of computing a mapping S from a poset P to a set of triple L such that the number of time steps needed to carry out all the operations in the poset is a minimum subject to these constraints (apart from other architectural constraints).

1) *forall i such that* $S(o_i) = l \in L$, *l.t* $\geq$ *max(w(e_j)) for all j* $\in$ *ES(o_i).*
2) *RT(l.r)* $\in$ *OR(o_i)*
3) *C(l.r)=l.c*
4) $\forall$ *t* $\forall$ *c* $\forall$ *rt* $\sum$ *(l_i.t = t)* $\wedge$ *(l_i.c = c)* $\wedge$ *(RT(l_i.r) = rt)* $\leq$ *N(c,rt)*
5) $\forall$ *t* $\forall$*i* $\forall$*j NT(t,ci,cj)* $\leq$ *BW(ci,cj)*

---

input: A dataflow graph
output: A triple (slot,cluster,unit) for each node in G
var
$ready\_list$: priority queue of nodes (instructions)
i : instruction under consideration
$cluster\_list$: priority queue of clusters
c : cluster under consideration
fu_list : priority queue of functional units
u : functional unit under consideration
ready_list.init(G)
while (!$ready\_list.is\_empty()$) **do**
  while (!$readylist.allTried()$) **do**
    $i \Leftarrow ready\_list.dequeue()$
    $cluster\_list.init(i)$
    $c \Leftarrow cluster\_list.dequeue()$
    $fu\_list.init(i, c)$
    $u \Leftarrow fulist.dequeue()$
    $schedule(i, c, u))$
    if $explicitMvNeeded(i, c, u))$ **then**
      $scheduleExplicitMv(i, c, u)$
    end if
    mark i as scheduled on unit u in cluster c
  end while
  readylist.update(G)
  advanceCycle()
end while

Fig. 2. Integrated spatial and temporal scheduling algorithm

3. The Algorithm

Our algorithm extends the standard list scheduling algorithm and includes following steps.

1) Prioritizing the ready instructions
2) Assignment of a cluster to a selected instruction
3) Assignment of a functional unit in the chosen cluster to a selected instruction

Figure 2 gives a very high level description of our algorithm. In what follows we will describe how each of these steps are performed in our algorithm.

3.1. Prioritizing the Nodes

Nodes in the ready list are ordered by a three component ordering function. Before describing our ordering function we introduce a term that we will use in defining our cost function.

Horizontal mobility: Horizontal mobility of an instruction is defined as the number of different functional units that are capable of performing the operation. In a VLIW architecture, horizontal mobility of an instruction is indicative of the freedom available in scheduling an instruction in a particular cycle. An instruction which is specific about a particular functional unit should be given priority over one with many possible scheduling alternatives. Horizontal mobility of an instruction is a function of a particular architecture implementation.

The three components of ordering function are as follows.

1) Vertical mobility (ALAP - ASAP, ALAP (ASAP) is the latest (earliest) possible schedule for an instruction)
2) Horizontal mobility (number of different functional units capable of performing this operation)
3) Number of consumers (number of successor instructions dependent on this instruction in the DFG)

The rationale behind using this three-component ordering function is as follows. The instructions with less vertical mobility should be scheduled early and are given priority over instruction with more vertical mobility to avoid unnecessary stretching of the schedule. Instructions with the same vertical mobility are further ordered in decreasing order of their horizontal mobility. The instruction which is very specific about a particular resource is given preference over others. An instruction with more number of successors is more constrained in the sense, its spatial and temporal placement affects the scheduling of more number of instructions and hence should be given priority.

3.2. Cluster Assignment

Once an instruction has been selected for scheduling we make a cluster assignment decision. The primary constraints are

- The chosen cluster should have at least one free resource of the type needed to perform this operation

- It should be possible to satisfy the communication needs for the operands of this instruction on the cluster either by snooping the operands or by scheduling explicit MV instructions in the earlier cycles given the bandwidth of the channels among clusters and their usage.

Selection of a cluster from the set the feasible clusters is done primarily based on communication cost metric given below.

$$comm\_cost = A * current\_comm + B * future\_comm + C * explicit\_mv \quad (1)$$

where

current_comm	Number of operands snooped in the current cycle
future_comm	Number of future communications required due to this binding
explicit_mv	Number of explicit MV operations in the earlier cycles

The communication cost models proposed here is generic enough to handle different ICC models or even a hybrid communication model where communication among clusters is possible by combination of different ICC models. The communication cost is computed by determining the number and type of communication needed by a binding. *explicit_mv* take care of communications that can be due to non-availability of snooping facility in a particular cycle (because of cross-path saturation) or in the architecture itself. Only those clusters which have enough communication slots and required resources for scheduling MV in the earlier cycles are considered. *current_comm* is determined by number of operands that reside on a cluster other than the cluster under consideration and can be snooped using the snooping facility available in current cycle. *future_comm* is predicted by determining the successors of this instruction which have one of their parents bound on a cluster different from the current one. In case some of the parents of some successor are not yet bound, the calculation is done assuming that it can be bound to any of the clusters with equal probability. We found that A=0.5, B=0.75 and C=1.0 works well in practice. These values are for an architecture that has limited snooping facility available. Since *explicit_mv* is a pure overhead in terms of code size, register pressure as well as resource usage, it is assigned double the cost of the *current_comm* to discourage these explicit MVs and favors snooping possibility in the cost model proposed. *future_comm* is also assigned a lesser cost optimistically assuming that most of them will be accommodated in the free-of-cost communication slot. The values of different constants can be changed to reflect the ICC model under consideration. Thus the cost function is generic and require slight tweaking for working with an architecture accommodating a different ICC model. In case of a tie in the communication cost, the instruction is assigned to a cluster where it can be scheduled on a unit least needed in current cycle with the same communication cost (further discussed in next subsection).

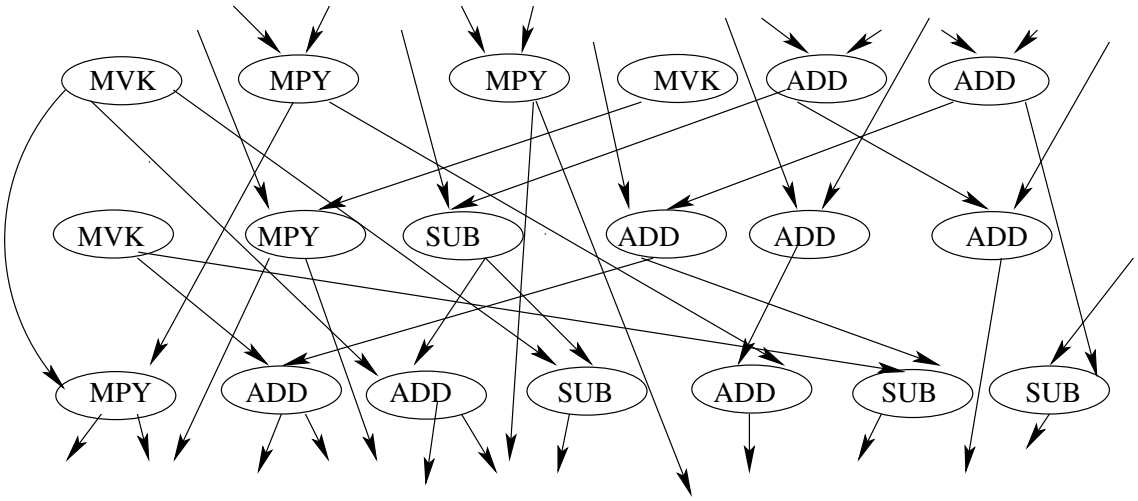

Fig. 3. Partial DFG of a media Application

3.3. Functional Unit Binding

Our algorithm is designed for a general machine model where an instruction can be executed on more than one functional unit but some functional units have some specialize task delegated to them. Resources may also differ in terms of their communicative capabilities. The operand snooping capability of a resource can be specified in terms of which operand a functional unit can read from which cluster. This necessitates doing an effective binding of functional units to instructions as well.

We maintain a resource need vector which contains the resource requirements of the instructions currently in the ready list. This vector is updated with change in the ready list and is used as a metric for effective functional unit binding. We select a functional unit which is least needed in the current cycle for the instructions in the ready list and assign this unit to the instruction under consideration. This avoids the situation where an instruction cannot be scheduled in the current cycle because a specialize resource needed by an operation later in the ready list is not available because of naive decision taken earlier. Commutativity of operations such as ADD and MPY is exploited to preserve a resource with scarce communicative capability as far as possible for prospective non-commutative instruction. Instructions with same vertical mobility value but different horizontal mobility value are presented in increasing order of horizontal mobility by the ordering function to help binding of as many instructions as possible in the current cycle. Since some instructions may have low vertical mobility, they are presented earlier for scheduling consideration despite their high horizontal mobility. Due to limitations on operand snooping capabilities of functional units, an instruction may require snooping a particular operand while none of the free functional units available has capability to snoop the desired operand. Our algorithm handles such

situations by shuffling functional units already bound to instructions in the current cycle and free units available. If possible we bind an instruction which is already scheduled to alternate units and make a unit free with the desired capability for the instruction under consideration.

It may be noted that a blind allocation of functional units may lead to stretched schedules. A blind allocator may allocate a unit with some specialized task to an operation which can be scheduled on an alternate free unit.

3.4. Scheduler Description

Initially the DFG is preprocessed and various DAG properties like height, depth and mobility of each of the node are computed. This is followed by initialization of the ready list with the root nodes of the graph. The ready list also has an associated resource vector which is used to keep track of the approximate resource requirements of the nodes currently in the ready list. The enqueue method of ready list impose a priority on nodes using a three component priority function described above. The dequeue method simply returns the next untried node (operation) from the priority ordered ready list and this node is then used to initialize a priority list of clusters on which instruction can be scheduled in the current cycle. Dequeue method of prioritized cluster list returns the highest priority cluster. The selected cluster and the node are then used to select feasible functional units on the cluster for scheduling the instruction under consideration. The dequeue method of the functional unit list returns the least needed resource in current cycle for binding the instruction. Any explicit MV operation if needed in an earlier cycle is now scheduled. The cluster initialization routine guarantees that the free slot for an explicit MV operation if needed, is available before putting the cluster in the feasible cluster list. If an operation cannot be scheduled on any cluster the operation is marked as tried and the next operation is considered. Once all the operations have been considered the cycle counter is advanced and ready list and the resource need vector are updated with operations which are now ready for scheduling.

3.5. An Example

In this sub section we present an example to demonstrate the benefit of integrating scheduling decisions for extended operand clustered processors to achieve high performance without severe code size penalty. Figure 3 is a partial DFG typical of a DSP application. We have shown only parts of the full DFG and the scheduled code due to space limitations. Figure 4, 5 and 6 present the schedule generated by our integrated scheduler, a possible schedule using UAS [17]. and schedule possible using the pre-partitioning scheme [13]. respectively.

We consider the TMS320C6X architecture. The maximum number of instructions that can be executed in a cycle are 8 and the hardware provides 1 communication in each direction per cycle without any

```
1.1MVK .S1 8, VR109
1.2||MPY .M2 VR70, VR90, VR1231
1.3||MPY .M1 VR2239, VR1260, VR1243
1.4||MVK .S2 9, VR147
1.5||ADD .L1X VR1260, VR1195, VR65
1.6||ADD .L2X VR106, VR70, VR102

2.1MVK .S2 12, VR171
2.2||MPY .M2 VR1254, VR147, VR1255
2.3||SUB .L1 VR65, VR67, VR108
2.4||ADD .D2 VR102, VR103, VR107
2.5||ADD .S1X VR1230, VR1254, VR89
2.6||ADD .D1 VR1242, VR65, VR95

3.1MPY .M2X VR1231, VR109, VR1273
3.2||ADD .D2 VR107, VR171, VR114
3.3|| ADD .D1 VR109, VR108, VR115
3.4|| SUB .L1 VR109, VR108, VR116
3.5|| ADD .S1X VR89, VR1231, VR100
3.6|| SUB .S2 VR107, VR171, VR119
3.7|| SUB .L2 VR103, VR102, VR118
```

Fig. 4. Scheduled DFG using Integrated Algorithm

```
x: MV .L2X VR1242, VR2854[a]

1.1MVK .S1 8, VR109
1.2||MPY .M2 VR70, VR90, VR1231
1.3||MPY .M1 VR2239, VR1260, VR1243
1.4||MVK .S2 9, VR147
1.5||ADD .L2X VR1260, VR1195, VR65
1.6||ADD .L1X VR106, VR70, VR102

2.1||MV .L1X VR1254, VR2843
2.2||MV .L2X VR102, VR2871

3.1MVK .S2 12, VR171
3.2||MPY .M2 VR1254, VR147, VR1255
3.3||SUB .L1X VR65, VR67, VR108
3.4||ADD .L2X VR102, VR103, VR107
3.5||ADD .S1 VR1230, VR2843, VR89
3.6||ADD .D2 VR2854, VR65, VR95

4.1MPY .M2X VR1231, VR109, VR1273
4.2||ADD .D2 VR107, VR171, VR114
4.3|| ADD .D1 VR109, VR108, VR115
4.4|| SUB .L1 VR109, VR108, VR116
4.5|| ADD .S1X VR89, VR1231, VR100
4.6|| SUB .S2 VR107, VR171, VR119
4.7|| SUB .L2 VR103, VR2871, VR118
```

Fig. 5. Scheduled DFG using UAS-MWP Algorithm

[a]Scheduled in an earlier cycle

```
1.1MVK .S1 8, VR109
1.2||MPY .M2 VR70, VR90, VR1231
1.3||MPY .M1 VR2239, VR1260, VR1243
1.4||MVK .S2 9, VR147
1.5||ADD .L1X VR1260, VR1195, VR65
1.6||ADD .L2X VR106, VR70, VR102

2.1MVK .S2 12, VR171
2.2||MPY .M2 VR1254, VR147, VR1255
2.3||SUB .L1 VR65, VR67, VR108
2.4||ADD .D2 VR102, VR103, VR107
2.5||ADD .S1X VR1230, VR1254, VR89
2.6||ADD .D1 VR1242, VR65, VR95

3.1|| MV .L2X VR89, VR149
3.2|| MV .L1X VR103, VR150

4.1MPY .M2X VR1231, VR109, VR1273
4.2||ADD .D2 VR107, VR171, VR114
4.3|| ADD .D1 VR109, VR108, VR115
4.4|| SUB .L1 VR109, VR108, VR116
4.5|| ADD .L2 VR149, VR1231, VR100
4.6|| SUB .S2 VR107, VR171, VR119
4.7|| SUB .S1X VR150, VR102, VR118
```

Fig. 6. Scheduled DFG using Pre-partitioning Algorithm

extra cost. Assume that the earlier scheduling decisions were such that instructions producing VR2239, VR1260, VR106, VR67, VR1242 and VR1230 are bound to cluster 1 while instructions producing VR70, VR90, VR1195, VR103 and VR1254 are bound to cluster 2.

Our scheme tries to schedule as many instructions as possible in the current cycle while minimizing the need for communication in the current as well as future cycles. Wherever possible, free-of-cost communication is used to exploit the available ILP. Commutativity of operations such as ADD and MPY is exploited wherever possible to combat the limitations in snooping capability of functional units. The ordering function proposed together with the systematic functional unit binding mechanism enable packing of as many instructions as possible in each cycle.

The pre-partitioning scheme [13] tries to assign clusters in such a way as to reduce the communication while trying to balance the load among clusters. Finally, our temporal scheduler is used to map functional units to the instructions in the cluster assigned by the partitioner. In our implementation of the pre-partitioning scheme we prefer to snoop operands wherever possible to assure fairness in comparison with our integrated approach.

UAS algorithm with MWP heuristic for cluster selection [17] assigns an instruction to a cluster where most of the predecessors of the instruction reside. However the MWP heuristics does not take into account the future communication cost of a binding. UAS algorithm as proposed by Ozer et el. does not propose

any particular priority order for considering instructions. There is also no notion of functional unit binding in the scheme proposed by Ozer et al.

Instructions 1.5 and 1.6 have their operands on alternate clusters. They are *equally likely* to be assigned to either of the clusters because UAS does not consider the future communication cost of a binding. Let us assume in the worst case that these instructions are assigned to clusters 2 and 1 respectively and that they are finally bound to units L2 and L1 as shown in the figure 5. The cycle is advanced to 2 because there are no more instructions to be considered in the current cycle. Instructions 2.3 and 2.4 (3.3 and 3.4 in figure 5)[1] in cycle 2 have their operands residing in the alternate clusters in the case of UAS. It may be noted that integrated algorithm has scheduled instructions 1.5 and 1.6 (figure 4) in cycle 1 in such a way that the instructions 2.3 and 2.4 (figure 4) can be scheduled without the need for cross-path. UAS does not consider future communication cost that may arise due to a binding and thus these instructions are scheduled using the cross-path. Instruction 2.5 (figure 4) and instructions 2.5 and 2.6 (3.5 and 3.6 in figure 5) in the same cycle again need to read one of their operand from another cluster in the case of integrated scheduler and UAS respectively. Integrated scheduler is able to schedule it using a free cross-path. However UAS requires explicit MV operations in the earlier cycles to schedule these instructions as the cross-path has already been used in scheduling instructions 2.3 and 2.4 (3.3 and 3.4 in figure 5). Required explicit MVs can not be scheduled in the cycle 1 as cross-path is already in use in the cycle 1. Let us assume that VR1242 can be moved from cluster 2 to cluster 1 in one of the free communication slot available in an earlier cycle (say x). Due to unavailability of appropriate cross-path or resource in any of the earlier cycles (after the cycle VR1254 is produced) the other MV operation is scheduled in the separate cycle and the current cycle becomes cycle 3. As in figure 5 UAS could accommodate 6 instructions in the cycle 3 at the cost of two explicit MV operations (x and 2.1) and an extra cycle (2) to accommodate a MV instruction. Instruction 4.7 (figure 5) in the case of UAS again need the cross-path to read VR102 from cluster 1 in the cycle 4. However the cross-path is already in use in this cycle as well as in the cycle 3. Since VR102 is produced in the cycle 1, an explicit MV operation is required to schedule this instruction in the current cycle. This explicit move operation can be accommodated in the extra cycle (number 2) introduced earlier. The value of VR102 is moved to VR2871 and the current instruction is modified to use VR2871 instead of VR102. This simple example demonstrates that the schedule generated using UAS suffers from several drawbacks like poor performance, low functional unit utilization and increased register pressure and code size due to more number of explicit MV operations. It may be noted that the integrated scheduler is able to schedule all the instruction as shown in the figure

[1]Instructions are actually scheduled as in figure 5, but for the sake of explanation, let us assume that these instructions 3.3 and 3.4 (similarly throughout the example) are scheduled in cycle 2, as instruction 2.3 and 2.4 respectively.

4 without the need for any explicit MV operation.

The first two cycles in case of schedule generated by pre-partitioning scheme have the same schedule as the integrated scheme in this example. However the difference occurs in the third cycle. To balance the load among clusters, the pre-partitioning scheme chooses to assign the instruction 4.5 (ADD VR149, VR1231, VR100) to cluster 2 because, communication cost is the same for both the clusters. The temporal scheduler fails to assign this instruction in an earlier cycle due to reasons that follow. While assigning functional units to instructions in cycle 3, the temporal scheduler finds that the cross path already in use by an earlier MPY instruction. So This instruction (i.e., instruction 4.5) cannot snoop one of its operands from the other side. The temporal scheduler tries to schedule an explicit MV in one of the earlier cycles, but is not successful in this case because VR89 is produced only in the previous cycle (cycle 2). The required MV (instruction 3.1) is scheduled in a separate cycle (cycle 3 in the case of the pre-partitioning scheme) and the cycle under consideration becomes the cycle 4 thereby pushing instruction 4.5 to cycle 4. The instruction is now modified to use VR149 instead of VR89 and is scheduled on unit L2. The next instruction (4.6) is assigned to cluster 2 and the temporal scheduler is able to schedule it on unit S2 in the current cycle.

It is not possible to schedule instruction 4.7 on cluster 2 because this overloads cluster 2. There is no free resource that can be used to schedule this instruction. A Spatial scheduler which is more aggressive towards balancing the load rather then minimizing communication may assign it to cluster 1 (as shown in the figure 5) but this will require at least one explicit MV operation as well as snooping an operand in the current cycle (i.e., cycle 4) and will incur overheads associated with the explicit MV operation. This MV (instruction 3.2) cannot be scheduled in the first two cycles as the cross path is already in use in these cycles and is scheduled in the cycle 3. The instruction 4.7 is modified to use VR150 instead to VR103. In any case the schedule is stretched by at least one cycle and incurs the overheads associated with two explicit MV operations in this simple example .

An integrated schedulers knows the instructions which are ready to be scheduled in the current cycle, their resource and cross path needs as well as cross path usage in current and earlier cycles. Exploiting this knowledge, our integrated scheduler is able to schedule all the instructions in three cycles in this example. The integrated scheduler uses the free-of-cost cross path facility to the maximum extent possible to get a better performance without too many explicit MV operations. A pre-partitioning algorithm striving to minimize communication can overload a cluster while the one aggressive towards exploiting ILP can generate more communication than what the available bandwidth can accommodate in a cycle.

An integrate scheme like UAS does not consider the future communication that may arise due to a binding and this may lead to stretched schedule because of more communication than what the available

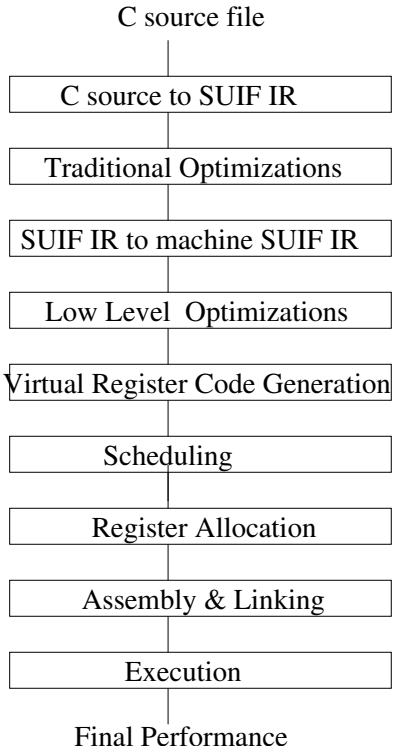

Fig. 7. Scheduling Framework

bandwidth can accommodate in a cycle and more number of explicit move operations. An explicit move operation reserves a resource for a cycle on the machine model under consideration and this may lead to a delay in scheduling other instructions, apart from the increase in register pressure and code size. Another drawback of UAS is due to the fact that it does not consider functional unit binding. However, effective functional unit binding is important in the case of the machine model under consideration because of variations in operational and communicative capabilities of functional units. Communication among clusters is thus not fixed but varies depending on functional unit to which an instruction is bound.

4. Experimental Evaluation

4.1. Setup

We have used the SUIF compiler [2] along with MACHSUIF library [1] for our experimentation. Figure 6 depicts the flow. We perform most of the optimizations on both high level and low level intermediate code. We generate code for TI TMS320C6X [23]. Ours is a hand-crafted code generator which is designed to take advantage of the specialized addressing modes of the architecture. Code generation is followed by a phase of peephole optimization and this highly optimized code is passed to our scheduler. The scheduler annotates instruction with the time-slot, cluster and functional unit information. Register allocation is performed on the scheduled code using priority based graph coloring [4]. Any spill code that is generated

is scheduled in separate cycles. After adding procedure prologue and epilogue, we emit the scheduled assembly code in a format acceptable to the TI assembler. After assembly and linking, we carry out simulation of the generated code using the TI simulator for the TMS320C6X architecture [23].

TABLE 1
DETAILS OF THE BENCHMARK PROGRAMS USED

Name	Description	Category	L/N
AUTOCORR	auto correlation	filters	.15
CORR 3x3	3x3 correlation with rounding	Filters	.11
FIR	Finite impulse response filter	Filters	.12
FIRCX	Complex Finite impulse response filter	Filters	.09
IIRCAS	Cascaded biquad IIR FILTER	Filters	.18
GOURAUD	Gouraud Shading	Imaging	.46
MAD	Minimum Absolute Difference	Imaging	.17
QUANTIZE	Matrix Quantization w/ Rounding	Imaging	.38
WAVELET	1D Wavelet Transform	Imaging	.16
IDCT	IEEE 1180 Compliant IDCT	Transform	.09
FFT	Fast fourier transform	Transform	.19
BITREV	Bit Reversal	Transform	.31
VITERBI	Viterbi v32 pstn trellis decoder	Telecom	.17
CONVDOC	Convolution Decoder	Telecom	.20

TABLE 2
LEGEND

Legends	Meaning
I	Integrated Algorithm
UC	Unified Assign and Schedule with Completion Weighted Predecessors Ordering
UM	Unified Assign and Schedule with Magnitude Weighted Predecessors Ordering
LP	Lapinskii's pre-partitioning Algorithm

4.2. Performance

Table 1 summarizes the key characteristics of benchmark programs we have used for experimental evaluation of our algorithm and a comparison with the pre-partitioning algorithm and UAS. These benchmarks are mostly unrolled inner loop kernels of MEDIABENCH [14], a representative benchmark of multimedia and communication applications and is specially designed for embedded applications. We have selected kernels from all the major embedded application areas. These include filter programs such as complex FIR, Cascaded IIR and autocorrelation, imaging routines such as wavelet transformation, quantization and shading and some transformation routines such as FFT and DCT. Some programs from the telecommunication domain like viterbi decoder, convolution decoder etc have also been used. Table 1 also mention the L/N ratio for each benchmark, where L is the critical path length and N is the number of nodes in the DFG for each kernel. L/N ration is an approximate measure of the available ILP in the program. When L/N is high, the partitioning algorithm has little room to improve the schedule. Programs

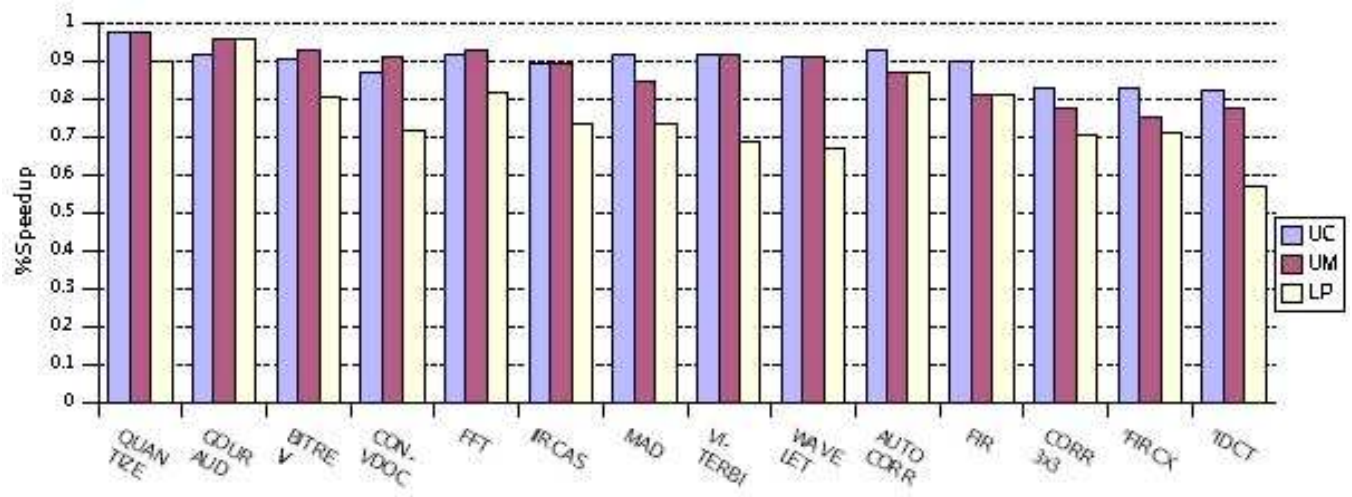

Fig. 8. Comparative Performance of Integrated, UAS and pre-partitioned scheduling

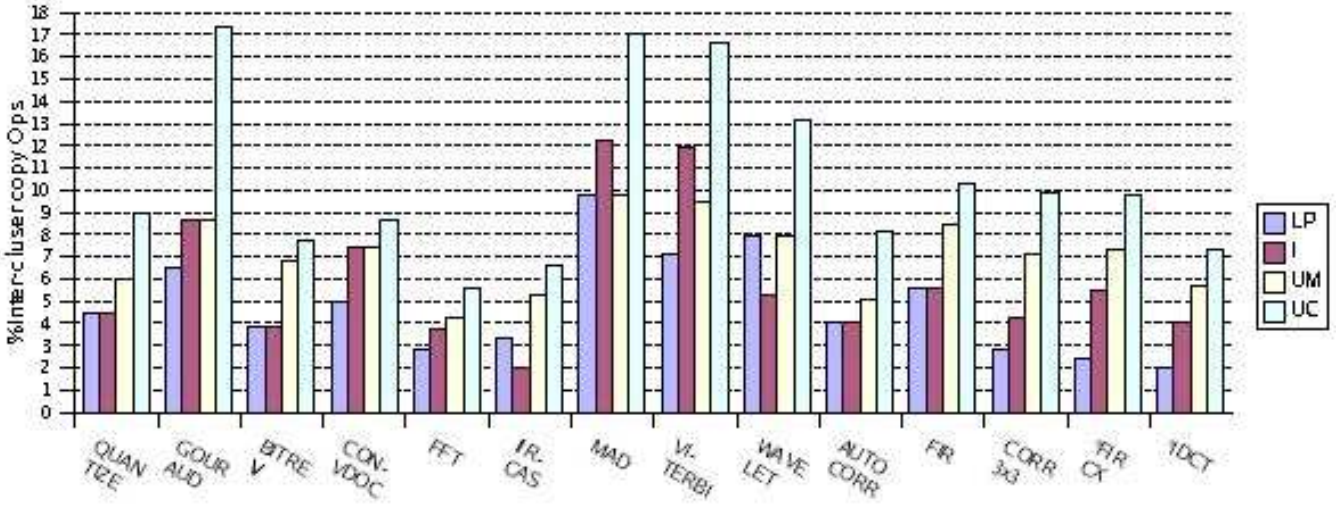

Fig. 9. %Distribution of explicit inter-cluster MV instructions vs. Other instructions

with low L/N exhibit enough ILP for a careful partitioning and effective functional unit binding algorithm to do well. The selected programs have L/N ratios varying between .09 to .46.

Figure 8 depicts the speed-up of UAS with MWP heuristic for cluster selection (UM), UAS with CWP heuristic for cluster selection (UC) and pre-partitioning scheme (LP) as compared to integrated algorithm (I). Integrated algorithm could attain significant speed-up over UC, UM and LP on program exhibiting high ILP (low L/N). Moderate and marginal speed-up is observed over programs with medium and little amount of ILP. As the amount of parallelism available in the program increases, the optimal schedule length is dictated by the resource constraints and effective partitioning and functional unit binding become more important.

Figure 9 presents the percentage distribution of explicit inter-cluster MV instructions vs. other instruc-

tions for I, UM, UC and LP. LP having a global view of DFG and intending to reduce ICC incurs the least code size penalty. UC incurs highest code size penalty. Number of MV instructions introduced in the case of UC becomes prominent in the case of high ILP routines. Integrated algorithm performs consistently better than UC and UM and slightly worse than LP in terms of number of explicit MV instructions used for ICC. We discuss the reasons for such a behavior in the next sub-section.

4.3. Discussion

In this section we discuss the performance of our integrated scheduling algorithm and our implementation of UAS and the pre-partitioning algorithm [13]. Integrated scheduling algorithm effectively exploits the following information available at schedule time.

1) Free cross-paths available in the current cycle as well as earlier cycles
2) Free functional units available in the current cycle as well as earlier cycles
3) Number and types of operations ready to be scheduled in the current cycle and their resource requirements

Given the precise knowledge of these facts our algorithm tries to schedule an operation in current cycle while minimizing the communication in the current as well as future cycles. Integrated scheme strives to bind an operation to a functional unit in a cluster where its operand can be snooped from the other cluster rather than explicitly using the communication path. This urge for avoiding explicit cross-cluster MV operations while maximizing utilization of the snooping capability helps to reduce the extra MV instructions and consequently the reservation of functional units for MV operation, code size as well as register pressure. Functional unit binding is done while taking care of other operations to be scheduled in current cycle and their needs for functional units and the cross path. This helps in combating the situation where earlier naive decisions lead to unnecessary delay in scheduling an operation in the current cycle and the consequent stretch of schedule

Our scheme is very aggressive in utilizing the free cross path and tries to schedule an instruction in the current cycle by trying to schedule communication in the earlier cycles using the free communication slot wherever necessary. Thus in some cases the integrated scheme compromises code size for performance. However the code size in most of the cases is the same as that of the pre-partitioning scheme and is only marginally more for some benchmarks. The primary reason is that the algorithm prefers to snoop operand from other clusters rather than simply inserting a MV operation.

An integrated scheme like UAS does not consider any future communication that may arise due to a binding and this may lead to a stretched schedule because of more communication than what the available bandwidth can accommodate in a cycle and hence more number of explicit move operations. An explicit

move operation reserves a resource for a cycle on the machine model under consideration and this may lead to poor resource utilization apart from increase in register pressure and code size. These are the reasons why UAS suffers from performance and code size penalty. The paper by Ozer at al. does not propose any particular order for considering instructions. However the order in which instructions are considered for scheduling has an impact on final schedule generated on clustered architectures in general and the machine model under consideration in particular. Another drawback of UAS is due to the fact that it does not consider functional unit binding. However, effective functional unit binding is important in the case of the machine model under consideration because of variations in operational and communicative capabilities of functional units. Communication among clusters is thus not fixed but depends on functional unit to which an instruction is bound.

Lapinskii's pre-partitioning algorithm suffers from the well known phase-ordering problem. We consider a fully pipelined machine model where each resource can initiate an operation every cycle. Resources vary in terms of their operational and communicative capabilities. Since on such a model, the resource and communication constraints are precisely known only while scheduling, any load balancing based on approximate schedule length leads to suboptimal schedule. Pre-partitioning algorithms have only approximate (or no) knowledge of above facts and thus the spatial scheduling decisions made by such an algorithm does not perform as well as integrated algorithm. To ensure the fairness in comparison in our implementation of Lapinskii's algorithm we give all possible scheduling benefit to their algorithm while honoring the cluster assignment decisions made by their algorithm. However, since their algorithm is mostly geared toward partitioning, some of the earlier decisions artificially constrains the temporal scheduler. Our Integrated algorithm does not suffer from these restrictions.

5. Previous Work

Recently there have been several proposals for scheduling on clustered VLIW architectures. As mentioned earlier there are two main approaches to scheduling on clustered VLIW architecture. Spatial scheduling followed by temporal scheduling and integrated spatial and temporal scheduling. Work in first direction is described in Gonzalez [3], Desoli [6] and Lapinskii [13] while the combined scheme is proposed by Leupers [15], Kailas [12] and Ozer [17]. In what follows we describe these approaches briefly. The work by Lapinskii et al. is described in some detail as we have compared our algorithm with their proposal.

Lapinskii et al. [13] have proposed an effective binding algorithm for clustered VLIW processors. Their algorithm performs spatial scheduling of instructions among clusters and relies on a list scheduling algorithm to carry out temporal scheduling. Instructions are ordered for consideration using an ordering function with the following components

1) As late as possible scheduling time of instruction
2) Mobility of instruction
3) Number of successors of an instruction

They compute the cost of allocating an instruction to a cluster using a cost function given as

$$icost(v,c) = fucost(v,c) * dii(v) * \alpha + buscost(v,c) * dii(BUS) * \beta + trcost(v,c) * lat(BUS) * \gamma \quad (2)$$

icost(v,c)	The cost of binding the instruction v to cluster c
fucost(v,c)	The functional unit serialization penalty
buscost(v,c)	The bus serialization penalty
trcost(v,c)	The data transfer penalty of scheduling v on cluster c.
dii(v)	The data introduction interval of the resource on which v is scheduled
dii(BUS)	The data introduction interval of the BUS
lat(BUS)	The latency of the BUS

Their cost function considers the load on the resources and buses to assign the nodes to clusters. The load on functional units and communication channels is calculated by adapting the force directed scheduling approach [18] of behavioral synthesis. They emphasize that their partition algorithm strives to exploit the parallelism while minimizing inter cluster communication is a secondary criterion. They have also proposed two binding improvement functions for further improving the schedule at some computational expense. These improvement functions reconsider nodes at the boundary of partition and look for opportunities for improvement.

Though they have proposed a good cost function for cluster assignment, as we have mentioned earlier, partitioning prior to the scheduling phase takes care of the resource load in an approximate manner. The exact knowledge of load on clusters and functional units is known only while scheduling. We believe that a scheme which performs scheduling along with partitioning can do a better job of load estimation and hence better binding of operations to clusters and resources in clusters. They have shown the performance of their algorithm based upon a statically determined schedule cycle. No Register allocation is done and the performance is given based on hypothetical cluster VLIW architecture configurations while assuming that all instructions are of unit latencies. There is no consideration of functional unit binding as their algorithm is geared towards partitioning only. They have compared their algorithm with the one proposed by Desoli et al [6] and have shown significant performance improvement.

Performance quoted here is more realistic as it is is based on running scheduled and register allocated code on a real clustered VLIW processor. We are in the process of adapting a simulator for various clustered VLIW configurations and performance measurement based on that.

Gonzalez et al. [3] have proposed a graph partitioning based approach that is used as a pre-partitioning phase to modulo scheduling. In their approach a multi-level graph partitioning approach is used to bind the instructions of a DFG to clusters. The first phase is graph coarsening in which a maximal edge weight matching algorithm is used to find a graph matching and collapsing the match nodes into one. This is repeated until the number of instructions are same as the number of clusters. This is followed by the partition refining phase that considers possible movement of instructions among clusters with the following objectives.

- Reducing the load on an overloaded resource in one cluster while not overloading the resource under consideration in another cluster
- Benefit in execution time
- Increasing the slack of edges
- Reducing number of edges between clusters

This work is geared more towards finding a partition for cyclic regions so as to reduce the initiation interval and get a better modulo schedule. They also perform various loop transformations to improve performance. It may be interesting to adapt this scheme to extended operand processors but we are not aware of any work in this direction.

Desoli and Faraboschi [6] have proposed an elegant algorithm for spatial scheduling called partial component clustering (PCC). The algorithm has two phases. In the first phase the DAG to be scheduled is partitioned into several components using a depth-first search algorithm and a maximum component size threshold. These components are then clusterized while striving for minimum ICC and balancing load among clusters. The second phase is characterized by improving the initial binding of instructions using an iterative algorithm to reduce the schedule length or ICC. The major objective of this algorithm is to minimize the communication among clusters.

This algorithm will perhaps perform well for clustered architectures where the communication among clusters is possible only through separate MV instructions. In the case of extended operand clustered VLIW architectures mere communication reduction may not produce the optimal schedule because the emphasis should be on exploiting free communication slots to map the application ILP to hardware.

Next we briefly discuss some schemes which try to integrate spatial and temporal scheduling in a single framework and contrast these schemes with ours.

Ozer et al. [17] have proposed an algorithm called unified-assign-and-schedule (UAS) that perform the combined job of partitioning and scheduling. UAS extends the list scheduling algorithm with cluster assignment while scheduling. After picking the highest priority node, it considers clusters in some priority order and checks if the operation can be scheduled in a cluster along with any communication needed due to

this binding. They have proposed various ways of ordering clusters for consideration such as, no ordering, random ordering, magnitude weighted predecessors (MWP) and completion weighted predecessors (CWP). MWP ordering considers the number of flow-dependent predecessors assigned to each cluster for operation under consideration. An instruction can be assigned to a cluster having the majority of predecessors. CWP ordering assigns to each cluster a *latest ready time* for operation under consideration depending on the cycle in which the predecessor operations produce the result. The clusters which produce the source operand of the operation late are given preference over others. The results show significant performance improvement over the bottom up greedy (BUG) algorithm on benchmark programs selected from SPECINT95 and MEDIABENCH. An ideal hypothetical cluster machine model has been used and the performance is based on statically measured schedule length.

To assure fairness in comparison, we have modified UAS to give it the benefits of an extended operand ICC model. To get the benefit of limited free-of-cost communication facility available in extended operand ICC model, we snoop the operand wherever possible to avoid unnecessary MV operations.

Leupers [15] proposed an algorithm for integrated scheduling of clustered VLIW processors. They use simulated annealing, a heuristic search algorithm for binding followed by scheduling algorithm. Partitioning is repeated with random rebinding of node and rescheduling in search of improved scheduled. The major problem with this algorithm is the large compilation time attributed to simulated annealing (even with two clusters).

The major difference between their algorithm and ours is that we orchestrate the binding using heuristic to utilize free communication facility in each slot to exploit ILP than spending time in random search to obtain better result. Their paper does not throw light on how the functional unit binding is performed. They have also reported the performance for compute intensive DSP kernels similar to ours.

Kailas et al. [12] have proposed an algorithm for combined binding scheduling and register allocation for clustered VLIW processors. Their greedy algorithm binds a node to the cluster where it can be executed at the earliest. This may lead to high ICC in the future while scheduling successor of this node due to unavailability of a communication slot to schedule the required MV and this may stretch the schedule as well. They also propose an on the fly register allocation. Insertion of spill code and scheduling of spill code is also integrated in main algorithm.

In general the integrated schemes proposed earlier mostly target architectures with explicit MV operations. Though these schemes can be modified for extended operand processors, we are not aware of any work in this direction. These schemes also do not consider the functional unit binding which becomes very important for extended operand processors because of the difference in operand snooping capabilities of different functional units. Targeting any of these schemes for a realizable clustered architecture requires

major variations to achieve optimal performance.

6. Conclusions and Future directions

We proposed a pragmatic integrated spatial and temporal scheduling algorithm for extended operand clustered VLIW processors with a systematic mechanism for cluster and functional unit binding. We have experimentally demonstrated a performance improvement of about 24% and 11% on the average as compared to a state-of-art phase-decoupled-algorithm and an earlier phase-coupled-algorithm respectively for a production chip. The algorithm does not suffer from the problem of excessive code size penalty to attain this performance improvement as the scheduler proposed effectively exploits the hardware operand snooping facility to minimize the need for explicit cross-cluster MV operations.

We propose following extensions to this work

1) Performance measurement for various clustered VLIW configurations. We are in the process of adapting a public domain simulator for various clustered configurations and ICC models.
2) Adapting the algorithm for software pipelining of inner loops.
3) New heuristics to reduce code size.

7. Acknowledgments

We wish to express our gratitude to Glenn Holloway of MACHSUIF team for prompt replies to our queries during the implementation of our scheduling framework. We also thank M. V. Panduranga Rao for fruitful discussions.

References

[1] MACHINE SUIF. http://www.eecs.harvard.edu/hube/software/software.html.

[2] SUIF Compiler System. http://suif.stanford.edu/.

[3] A. Aleta, J. M. Codina, J. Snchez, and A. Gonzlez. Graph-partitioning based instruction scheduling for clustered processors. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 150–159. IEEE Computer Society, 2001.

[4] F. C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, 12(4):501–536, 1990.

[5] J. Derby and J. Moreno. A high-performance embedded DSP core with novel SIMD features. In *Proceedings of the 2003 International Conference on Acoustics, Speech, and Signal Processing*, 2003.

[6] G. Desoli. Instruction assignment for clustered VLIW DSP compilers: A new approach. Technical Report, Hewlett-Packard, February 1998.

[7] P. Faraboschi, G. Brown, J. A. Fisher, and G. Desoli. Clustered instruction-level parallel processors. Technical report, Hewlett-Packard, 1998.

[8] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 203–213. ACM Press, 2000.

[9] J. Fridman and Z. Greefield. The tigersharc DSP architecture. *IEEE Micro*, pages 66–76, Jan. 2000.

[10] R. Ho, K. Mai, and M. Horowitz. The Future of Wires. *Proceeding of IEEE*, April 2001.

[11] T. I. Inc. Tms320c6000: a high performance dsp platform. http://www.ti.com/sc/docs/products/dsp/c6000/index.htm.

[12] K. Kailas, A. Agrawala, and K. Ebcioglu. CARS :A new code generation framework for clustered ILP processors. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA'01)*, January 2001.

[13] V. S. Lapinskii, M. F. Jacome, and G. A. De Veciana. Cluster assignment for high-performance embedded VLIW processors. *ACM Trans. Des. Autom. Electron. Syst.*, 7(3):430–454, 2002.

[14] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335. IEEE Computer Society, 1997.

[15] R. Leupers. Instruction scheduling for clustered VLIW DSPs. *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (Philadelphia, PA)*, October 2000.

[16] D. Matzke. Will physical scalability sabotage performance gains. *IEEE Computer*, 30(9):37–39, September 1997.

[17] E. Ozer, S. Banerjia, and T. M. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 308–315. IEEE Computer Society Press, 1998.

[18] P. G. Paulin and J. P. Knight. Force-directed scheduling in automatic data path synthesis. In *24th ACM/IEEE conference proceedings on Design automation conference*, pages 195–202. ACM Press, 1987.

[19] G. G. Pechanek and S. Vassiliadis. The manarray embedded processor architecture. In *Proceedings of the 26th. Euromicro Conference*, pages 348–355, 2000.

[20] S. Rixner, W. J. Dally, B. Khailany, P. R. Mattson, U. J. Kapasi, and J. D. Owens. Register organization for media processing. In *Proceedings of the Sixth International Symposium on High Performance Computer Architecture*, pages 375–386, 2000.

[21] N. Seshan. High VelociTI Processing. *IEEE Signal Processing Magazine*, March 1998.

[22] A. Terechko, E. L. Thenaff, M. Garg, J. van Eijndhoven, and H. Corporaal. Inter-cluster communication models for clustered VLIW processors. In *Proceedings of Symposium High Performance Computer Architectures*, February 2003.

[23] Texas Instruments Inc. TMS320C6000 CPU and Instruction Set reference Guide. http://www.ti.com/sc/docs/products/dsp/c6000/index.htm, 1998.