An Efficient Heuristic for Instruction Scheduling on Clustered VLIW Processors

Xuemeng Zhang Hui Wu Jingling Xue

School of Computer Science and Engineering
The University of New South Wales
{xuemengz, huiw, jingling}@cse.unsw.edu.au

ABSTRACT

Clustering is a well-known technique for improving the scalability of classical VLIW processors. A clustered VLIW processor consists of multiple clusters, each of which has its own register file and functional units. This paper presents a novel phase coupled priority-based heuristic for scheduling a set of instructions in a basic block on a clustered VLIW processor. Our heuristic converts the instruction scheduling problem into the problem of scheduling a set of instructions with a common deadline. The priority of each instruction v_i is the $l_{max}(v_i)$-successor-tree-consistent deadline which is the upper bound on the latest completion time of v_i in any feasible schedule for a relaxed problem where the precedence-latency constraints between v_i and all its successors, as well as the resource constraints are considered. We have simulated our heuristic, UAS heuristic and Integrated heuristic on the 808 basic blocks taken from the MediaBench II benchmark suite using six processor models. On average, for the six processor models, our heuristic improves 25%, 25%, 33%, 23%, 26%, 27% over UAS heuristic, respectively, and 15%, 16%, 15%, 9%, 20%, 8% over Integrated heuristic, respectively.

Categories and Subject Descriptors: C.1.1[Processor Architectures]: Single Data Stream Architectures - RISC / CISC, VLIW architecture; D.3.4 [Programming Languages]: Processors - Compilers; Optimisation

General Terms: Algorithms, Performance

Keywords: Clustered VLIW Processor, Instruction Scheduling, Inter-instructional Latency, Inter-cluster Communication Latency

1. INTRODUCTION

A Very Long Instruction Word (VLIW) processor consists of multiple functional units. All the functional units share a single register file. At each processor cycle, multiple instructions are issued and executed in parallel on different functional units. All the instructions are scheduled at compile time. Hence, an optimising compiler plays a key role to employ Instruction Level Parallelism (ILP) [1]. The objective of an optimising compiler is to schedule the instructions of a program such that the execution time of the program is minimised. As a result, the compiler needs to schedule as many instructions as possible at each cycle. Nevertheless, a large amount of ILP puts immense pressure on the processor's registers and bypass network.

A major problem with the classical VLIW processors is that a single register file hampers the scalability of the processor. Clustering is an efficient technique for improving the scalability of VLIW processors. In a clustered VLIW processor, the single register file is split into multiple register files with fewer registers and ports. Each cluster has its own functional units and local register file. Clusters are connected by an inter-cluster communication network [2].

Instruction scheduling for clustered VLIW processors becomes more challenging as there are additional inter-cluster communication constraints. If an instruction executed on a cluster needs the result produced by another instruction executed on a different cluster, the compiler needs to take into account the communication latency between these two clusters when scheduling these two instructions.

In this paper, we propose a novel priority based heuristic for scheduling a set of instructions in a basic block on a clustered VLIW processor. Our heuristic converts the instruction scheduling problem into the problem of scheduling a set of instructions with a common deadline. The priority of each instruction v_i is the upper bound on the latest completion time of v_i in any feasible schedule for a relaxed problem where the inter-instructional latency between v_i and all its successors, as well as the resource constraints are considered. Given a basic block, the time complexity of our heuristic is $O(ne)$, where n is the number of instructions in the basic block, and e is the number of edges in the precedence-latency graph. Our simulation results show that our heuristic performs significantly better than UAS heuristic [3] and Integrated heuristic [4].

2. RELATED WORK

The problem of scheduling a set of instructions in a basic block executed on a clustered VLIW processor such that the execution time of the basic block is minimised is NP-complete even if the processor has only one cluster [5]. A number of heuristics have been proposed. All the previous heuristics can be classified into two main categories, the phase decoupled approach and the phase coupled approach. The phase decoupled approach [6–8] partitions instructions into clusters before scheduling the instructions,

CASES'11, October 9–14, 2011, Taipei, Taiwan.

which suffers from the phase ordering problem as partitioning is performed without knowing the conflicts of resources and the final schedule. As a result, it may introduce unnecessary inter-cluster communication, and impose more constraints on the subsequent scheduling. The phase coupled approach [3, 4, 9–12] combines cluster assignment and scheduling into a single phase, which can avoid the phase ordering problem.

The Bulldog compiler [6] is the first study of instruction scheduling on a clustered VLIW processor. It uses a two-phase approach that separates cluster assignment from scheduling. The first pass assigns each instruction to a cluster using Bottom-Up Greedy (BUG) algorithm, and the second pass uses list scheduling [13] to construct a schedule that respects the cluster assignments. BUG uses the depth-first search and a latency-weighted depth priority scheme. It tries to put the instructions on the critical path on the same cluster so that inter-cluster data transfer latency can be minimised. Unfortunately this approach leads to the phase ordering problem as the BUG can not know the utilisation of functional units and inter-cluster data path. As a result, it can not fully utilise the machine resources to minimise the execution time of a basic block.

Lapinskii et al. [8] propose an effective binding algorithm for clustered VLIW processors. They use a three-component ranking function to determine the order in which each instruction is considered for binding:

1. As late as possible (ALAP) scheduling time of the instruction.
2. Mobility of the instruction, which is ALAP scheduling time of the instruction minus as early as possible (AEAP) scheduling time.
3. Number of successors of the instruction.

They compute the cost of binding an instruction to a cluster by a cost function that takes into account the data transfer penalty, resource and bus serialisation penalty. The problem with this algorithm is that the assignment of instructions to clusters prior to the scheduling phase can not get the exact information of load on resources and buses.

The Unified Assign and Schedule (UAS) heuristic [3] combines cluster assignment and scheduling into a single unified phase. The outer loop of the heuristic uses a list scheduler [13] to fill the instructions into the schedule cycle by cycle, and the inner loop considers each possible cluster assignment for each instruction. Each cluster is checked in priority order to see if it has a free functional unit for the instruction. Five different priority functions are investigated:

1. None: The cluster list is not ordered.
2. Random Ordering: The cluster list is ordered randomly.
3. Magnitude-weighted Predecessor (MWP): An instruction is placed into a cluster where the majority of its input operands reside.
4. Completion-weighted Predecessor (CWP): This gives priority to the cluster that will be producing source operand later.
5. Critical-Path in Single Cluster using CWP Heuristics (CPSC): The CWP heuristic is used but instructions on the critical path are forced to be assigned to the same cluster.

Once a cycle is scheduled, it will be never revisited. Therefore, the inter-cluster data movement can not be inserted into earlier cycles that are already scheduled, to reduce the delay of data movement and utilise the free resources.

Nagpal et al. [4] propose an effective heuristic that integrates temporal and spatial scheduling. The heuristic utilises the exact knowledge of available communication slots, functional units, and load on different clusters as well as future resource and communication requirements known only at schedule time. The heuristic extends the standard list scheduling algorithm and uses a three-component ordering function to order the ready instructions as follows:

1. Mobility of the instruction.
2. The number of different function units capable of executing the instruction.
3. Number of successor instructions.

Once an instruction is selected for scheduling, a cluster assignment decision is made as follows:

1. The chosen cluster should have at least one free resource of the type needed to perform this instruction.
2. A communication model is proposed to select a cluster based on the number of snoops in the current cycle, number of future communications required, as well as number of copy instructions in the earlier cycle.

A major problem with all the previous priority-based approaches is that they do not consider the processor resource constraints when computing the priority of each instruction. As a result, the priorities may be biased and may not reflect the relative importance of each instruction. Our heuristic is the first one that takes into account the processor resource constraints when computing the priority of each instruction.

There are a number of papers that study the problem of effectively scheduling loops on clustered VLIW processors by using software pipelining and modulo scheduling techniques [11, 12, 14–17]. The primary objective of software pipelining and modulo scheduling techniques is to efficiently overlap the different iterations of an inner-most loop.

3. INSTRUCTION SCHEDULING ON CLUSTERED VLIW PROCESSORS

3.1 Processor Model and Definitions

A target clustered VLIW processor has m identical clusters, C_1, C_2, ..., C_m. Each cluster has a set of functional units of different types. Each instruction can be executed only on a functional unit of the same type. The target processor is fully pipelined, i.e, the execution of each instruction takes only one processor cycle. For ease of descriptions, we assume that all the clusters are fully connected [1], i.e., the inter-cluster communication latency is a constant c between any two clusters. If instructions v_i and v_j are executed on different clusters and v_j needs the result of v_i, a copy instruction, executed on v_i's cluster, is required to transfer the result from v_i's cluster to v_j's cluster. The copy instruction, denoted by *copy*, also takes one cycle.

A basic block is represented by its precedence-latency graph which is a weighted DAG (directed acyclic graph) $G = (V, E, W)$, where $V = \{v_1, v_2, \ldots, v_n$: v_i is an instruction$\}$, $E = \{(v_i, v_j) : v_j$ is directly dependent on $v_i\}$, and $W = \{l_{i,j} : (v_i, v_j) \in E$, $l_{i,j}$ is the inter-instructional latency between v_i and $v_j\}$. An inter-instructional latency may exist between two instructions with a precedence constraint due to the pipelining architecture and off-chip memory latency. Fig. 1 is an example of a basic block with 12 instructions

[1]Our scheduling heuristic is also applicable to heterogeneous clusters and arbitrary communication network.

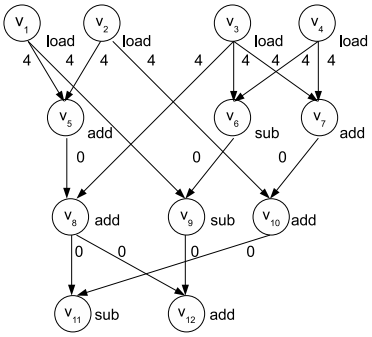

Figure 1: The weighted DAG

from v_1 to v_{12}.

The instruction scheduling problem studied in this paper is described as follows. Given a basic block represented by a weighted DAG $G = (V, E, W)$, find a valid schedule σ with minimum length satisfying the following constraints:

1. Resource constraints: 1) each instruction can be executed only on a functional unit of the same type; 2) At any time, only one instruction can be executed on each functional unit.

2. Precedence-latency constraints: For each $(v_i, v_j) \in E$, if v_i and v_i are on different clusters, $\sigma(v_i)+2+c+l_{ij} \leq \sigma(v_j)$ holds; otherwise, $\sigma(v_i) + 1 + l_{ij} \leq \sigma(v_j)$ holds.

Given two instructions v_i and v_j, if there is a directed path from v_i to v_j, then v_i is a predecessor of v_j and v_j is a successor of v_i. Especially, if $(v_i, v_j) \in E$, then v_i is an immediate predecessor of v_j and v_j is an immediate successor of v_i. If instruction v_i has no immediate predecessor, v_i is a source instruction. If instruction v_i has no immediate successor, v_i is a sink instruction. An instruction v_i is a sibling of an instruction v_j if v_j is an immediate predecessor of an immediate successor of v_i. Throughout this paper, all the successors of an instruction v_i are denoted by $Succ(v_i)$ and the number of elements in a set U is denoted by $|U|$.

Definition 1. *Given a DAG $G = (V, E)$ and an instruction $v_i \in V$, the successor tree of v_i is a directed tree $T(G, v_i) = (V', E')$, where $V' = \{v_i\} \cup Succ(v_i)$ and $E' = \{(v_i, v_j) : v_j \in Succ(v_i)\}$.*

In a weighted DAG G, if there is a directed path P_{ij} from v_i to v_j, the weighted path length of P_{ij} is the sum of its constituent edge weights and the number of instructions in P_{ij}, excluding the two end instructions v_i and v_j. The maximum weighted path length from v_i to v_j, denoted by l^+_{ij}, is the maximum of the weighted path lengths of all the paths from v_i to v_j.

Definition 2. *Given a weighted DAG $G = (V, E, W)$, an instruction $v_i \in V$ and an integer k, the k-successor tree of v_i is a subgraph $ST(G, k, v_i) = (V', E', W')$, where $V' = \{v_i\} \cup \{v_j : v_j \in Succ(v_i)\}$, $E' = \{(v_i, v_j) : v_j \in Succ(v_i)\}$ and each edge weighted l'_{ij} in W' is $l'_{ij} = l^+_{ij}$ if $l^+_{ij} < k$; otherwise, $l'_{ij} = k$.*

Fig. 2 is an example of 4-successor tree of v_3 in Fig. 1. Given a problem instance P: a set of instructions in a basic block represented by a weighted DAG and a clustered VLIW processor M, we define a new problem instance $P(d)$ as follows:

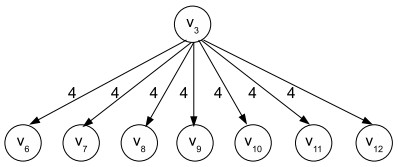

Figure 2: The 4-successor-tree of v_3

the same set of instructions as in P with a common deadline d and the same VLIW processor M. Conceptually, d can be any number. In order not to produce negative start time for any instruction, we select d to be $n * c * l_{max}$, where n is the number of instructions, c the inter-cluster communication latency, and l_{max} the maximum inter-instructional latency. A valid schedule for the problem instance $P(d)$ is a feasible schedule if the completion time of each instruction is not greater than the common deadline d.

Let $l_{max}(v_i)$ denote the maximum latency between v_i and all its immediate successors. Next, we define the $l_{max}(v_i)$-successor-tree-consistent deadline of an instruction v_i.

Definition 3. *Given a problem instance $P(d)$, the $l_{max}(v_i)$-successor-tree-consistent deadline of an instruction v_i, denoted by d'_i, is recursively defined as follows. If v_i is a sink instruction, d'_i is equal to the common deadline d; otherwise, d'_i is the upper bound on the latest completion time of v_i in any feasible schedule for the relaxed problem instance $P(v_i)$: a set $V' = \{v_i\} \cup Succ(v_i)$ of instructions with precedence and latency constraints in the form of the $l_{max}(v_i)$-successor tree $ST(G, l_{max}(v_i), v_i)$, deadline constraints $D' = \{d''_j : v_j \in V'$ and d''_j is the deadline of v_j in $P(v_i)$ and $(d''_j = d'_j$ if $j \neq i$ or $d''_j = d_j$ if $j = i)\}$, and the same clustered VLIW processor P. Formally, $d'_i = \max\{\sigma(v_i) : \sigma$ is a feasible schedule for $P(v_i)\}$.*

In our scheduling heuristic, the priority of each instruction v_i is its $l_{max}(v_i)$-successor-tree-consistent deadline which considers both precedence-latency constraints and resource constraints. As a result, it represents the relative importance of v_i more accurately than the deadline derived only from the precedence-latency constraints.

3.2 Computing the $l_{max}(v_i)$-Successor-Tree Consistent Deadline

Our heuristic consists of three major steps. Firstly, it schedules all the successors of the instruction v_i in the $l_{max}(v_i)$-successor tree as late as possible. If two instructions have the same deadline, the instruction with a larger latency will be scheduled first. Then, it finds the successor with the most impact on v_i's latest start time without considering inter-cluster communication latency. v_i will be tentatively scheduled on the same cluster as this successor in order to make v_i's start time as late as possible. Secondly, it computes the tentative latest start time of v_i by considering the communication latency between v_i and all the successors which are not on the same cluster as v_i. Thirdly, it tries to re-schedule the successors, which are not on the same cluster as v_i, on the same cluster as v_i to make v_i's start time later than its tentative start time.

Given two instructions v and w, let $L_{v,w}$ be the latency between v and w in the $l_{max}(v_i)$-successor tree. The pseudo

code of our algorithm for computing the $l_{max}(\ v_i)$-successor-tree-consistent deadline is shown as follows.

Algorithm *Computing the $l_{max}(v_i)$-successor-tree-consistent deadline*
Input *A weighted DAG G that represents a basic block;*
an array V of all the instructions;
a clustered VLIW processor M;
Output *The $l_{max}(V[i])$-successor-tree-consistent deadline of each instruction $V[i]$;*
begin
sort V in topological order;
for $i = |V| - 1$ *to* 0

/ Step 1 */*
if *$V[i]$ is a sink instruction*
the $l_{max}(V[i])$-successor-tree-consistent deadline of $V[i] = d$;
continue ;
construct the l_{max} $(V[i])$-successor tree T of $V[i]$;
let S be an array of $Succ(V[i])$ in T;
sort S in non-increasing order of deadlines;
sort all the instructions in S with the same deadline in non-increasing order of their latencies in T;
for $j = 0$ *to* $|S| - 1$
schedule $S[j]$ on an idle functional unit as late as possible;
$F[S[j]]$ = the start time of $S[j]$;

/ Step 2 */*
let a be an integer in $[0, |S| - 1]$ satisfying
$F[S[a]] - L_{S[a],V[i]} = \min\{F[S[j]]$
$-L_{S[j],V[i]} : j \in [0, |S| - 1]\}$;
let C' be the cluster on which $S[a]$ is scheduled;
let S' be the subset of $Succ(V[i])$ not scheduled on C';
$D' = F[S[a]] - L_{S[a],V[i]} - 1$;
for $c = 0, \cdots, |S'| - 1]$
$I[S'[c]] = F[S'[c]] - L_{S'[c],V[i]}$;
$D' = \min\{I[S'[c] - c - 2, D'\}$;

/ Step 3 */*
sort S' in non-decreasing order of I;
let p be the number of functional units on each cluster;
for $k = 0$ *to* $\min\{|S'| - 1, c * p\}$
/ check if $S'[k]$ can be moved to cluster C' */*
let A be an array containing $S'[k]$ and all the instructions scheduled on C';
sort A in non-increasing order of deadlines;
sort all the instructions in A with the same deadline in non-increasing order of their latencies in T;
for $j = 0$ *to* $|A| - 1$
schedule $A[j]$ on an idle functional unit of C' as late as possible;
$F[A[j]]$ = the start time of $A[j]$;
$t_1 = \min\{F[v_j] - L_{v_j,V[i]} - 1 : j = 0, \cdots, |A| - 1$
and v_j is scheduled on C' $\}$;
$t_2 = \min\{F[v_j] - L_{v_j,V[i]} - c - 2 : j = 0, \cdots, |A| - 1$
and v_j is not scheduled on C' $\}$;
if $\min\{t_1, t_2\} > D'$
move $S'[k]$ to C';
$D' = \min\{t_1, t_2\}$;
computing the latest start time $s[i]$ of $V[i]$;
the $l_{max}(V[i])$-successor-tree-consistent deadline of $V[i] = s[i] + 1$;
end

In order to maximise the start time of v_i, our algorithm checks if an instruction in S' can be moved to cluster C'. Next, we prove that at most the first $c * p$ instructions in S' need to be checked. Let t_{max} be the latest start time of v_i computed in Step 2. It is easy to see that the latest start time of v_i can be increased by at most c cycles. Therefore, at most the first $c * p$ instructions in S' need to be checked.

Example 1: Consider the basic block shown in Fig. 1. Assume that inter-cluster communication delay c is 2 cycles. The common deadline of all the instructions is $12 * 2 * 4 = 96$. The target clustered processor has two identical clusters C_1 and C_2. Each cluster has one ALU unit and one load/store unit. Assume that our algorithm has computed $d'_i(i = 6, 7, 8, 9, 10, 11, 12)$, where $d'_6 = d'_7 = d'_8 = 94$, $d'_9 = d'_{10} = 95$, and $d'_{11} = d'_{12} = 96$,. Next, we show how our algorithm computes the $l_{max}(v_3)$-successor-tree-consistent deadline d'_3. Recall that d'_3 is the upper bound on the latest completion time of v_3 in any feasible schedule for the relaxed problem instance $P(v_3)$, where the 4-successor tree of v_3 is shown in Fig. 2. Firstly, our algorithm schedules all the successors of v_3 as late as possible based on their $l_{max}(v_i)$-successor-tree-consistent deadlines and latencies in the relaxed problem instance $P(v_3)$. The schedule is shown in Table 1a. Secondly, our algorithm finds the tentative latest start time of v_3 with respect to the previous schedule and inter-instructional latencies as well as inter-cluster communication latency. The tentative latest start time of v_3 is 85, as shown in Table 1b. Lastly, our algorithm checks if $v_i(i = 8, 10, 12)$ can be moved to the cluster C_1 so that the resulting latest start time of v_3 can be increased. As a result, v_8 is moved to the cluster C_1, increasing the resulting latest start time of v_3 by one cycle. The final schedule for computing the $l_{max}(v_3)$-successor-tree-consistent deadline is shown in Table 1c. From the schedule shown in Table 1c, we get $d'_3 = 87$.

The $l_{max}(v_i)$-successor-tree-consistent deadline of each instruction v_i is shown in Table 2. For comparison, the priorities of all the instructions in Example 1 using AEAP scheduling time, ALAP scheduling time, instruction mobility, number of successors are also shown in Table 2.

Next, we analyse the the time complexity of our algorithm for computing all the $l_{max}(v_i)$-successor-tree-consistent deadlines.

THEOREM 1. *Given a basic block, the time complexity of our algorithm for computing all the $l_{max}(v_i)$-successor-tree-consistent deadlines is $O(ne)$, where n is the number of instructions in the basic block, and e is the number of edges in the precedence-latency graph.*

PROOF. The time complexity of one iteration of the outer **for** loop is dominated by the following parts:

1. Constructing the l_{max} $(V'[\mathrm{i}])$-successor tree T of $V'[i]$. It takes $O(e)$ time.
2. Sorting S. We can sort all the instructions of the basic block before the outer **for** loop. As a result, it takes $O(n)$ time to sort S.
3. The first inner **for** loop and the second inner **for** loop. Both **for** loops take $O(n)$ time.
4. The third inner **for** loop. Sortings take $\log n$ time as only one instruction is added to the set of instructions scheduled on cluster C'. As a result, it takes $O(n)$ time to check if the instruction $S'[k]$ can be moved to cluster C'. The maximum number of instructions to be checked, i.e., the number of iterations of the inner-most **for** loop, is $c*p$, where c is inter-cluster communication

Time	85	86	87	88	89	90	91	92	93	94	95	96
L/S on C_1												
ALU on C_1								v_6	v_7	v_9	v_{11}	
L/S on C_2												
ALU on C_2									v_8	v_{10}	v_{12}	

(a) Step 1

Time	85	86	87	88	89	90	91	92	93	94	95	96
L/S on C_1	v_3					*copy*						
ALU on C_1								v_6	v_7	v_9	v_{11}	
L/S on C_2												
ALU on C_2									v_8	v_{10}	v_{12}	

(b) Step 2

Time	85	86	87	88	89	90	91	92	93	94	95	96
L/S on C_1		v_3					*copy*					
ALU on C_1							v_8	v_6	v_7	v_9	v_{11}	
L/S on C_2												
ALU on C_2										v_{10}	v_{12}	

(c) Step 3

Table 1: Comptuting the $l_{max}(v_3)$-successor-tree-consistent deadline

Table 2: Priority of each instruction v_i

Instruction	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}
AEAP	0	0	0	0	5	5	5	6	6	6	7	7
ALAP	2	2	2	2	7	7	7	8	8	8	9	9
Instruction Mobility	2	2	2	2	2	2	2	2	2	2	2	2
Successor Number	2	2	3	2	1	1	1	2	1	1	0	0
$l_{max}(v_i)$-successor-tree-consistent Deadlines	88	88	87	87	93	94	94	94	95	95	96	96

latency and p is the number of functional units in each cluster. Since both c and p are very small, $c * p$ can be considered as a constant. Hence, this nested **for** loop takes $O(n)$ time.

Therefore, the time complexity of our algorithm is $O(ne)$. □

3.3 Instruction Scheduling Heuristic

In our scheduling heuristic, the priority of each instruction v_i is its $l_{max}(v_i)$-successor-tree-consistent deadline. A smaller deadline implies a higher priority. Our heuristic schedules instructions in non-decreasing order of their deadlines and tries to schedule the immediate predecessors of each instruction on the same cluster without delaying this instruction, in order to reduce the inter-cluster communication latency. Our heuristic works as follows:

At any time, it selects a ready instruction v_i with the smallest $l_{max}(v_i)$-successor-tree-consistent deadline. An instruction v_i is ready if both its inter-instructional latencies and inter-cluster communication latency have elapsed. Two cases are distinguished.

1. v_i has a sibling that is already scheduled. Let v_j be the immediate successor of v_i satisfying the following constraints: a) One immediate predecessor of v_j is already scheduled. b) v_j has the smallest $l_{max}(v_j)$-successor-tree-consistent deadline among all the immediate successors of v_i having an immediate predecessor already scheduled. Consider the following two cases:

   (a) There are two clusters C_s and C_t such that the start time of v_j is minimised if v_i is scheduled on either cluster C_s or cluster C_t, where the start time of v_j is the time at which v_j can be scheduled immediately after v_i is scheduled. In this case, schedule v_i on a cluster as early as possible.

   (b) Such clusters C_s and C_t do not exist. Schedule v_i on a cluster such that the start time of v_j is minimised, where the start time of v_j is the time at which v_j can be scheduled immediately after v_i is scheduled.

2. v_i is a sink instruction, or v_i does not have a sibling that is already scheduled. In this case, schedule v_i on a cluster as early as possible.

The reason that our scheduling heuristic distinguishes between Case (a) and Case (b) is that v_j may not be scheduled immediately after v_i and other instructions scheduled between v_j and v_i will affect the start time of v_j. In this case, scheduling v_i on a cluster earlier may make v_j start earlier. Consider Example 1 shown in the previous section. Assume our scheduling heuristic has constructed a partial schedule for instructions $v_k (k = 1, 2, \cdots, 8)$ as shown in Table 3a. Now consider scheduling v_9. v_9 has a sibling v_8 already been scheduled. If v_9 is scheduled on either C_1 or C_2, the start time of v_{12} is minimised. So this is where Case (a) is applicable. As we can see, the best choice is to schedule v_9 on cluster C_1 at cycle 8.

A schedule for Example 1 constructed by our heuristic is shown in Table 3a. For comparison, a schedule for Example 1 constructed by UAS heuristic with ALAP and CWP priority schemes [3] is shown in Table 3b, and a schedule for Example 1 generated by Integrated heuristic [4] is shown in Table 3c. We can see that our heuristic can effectively

Time	0	1	2	3	4	5	6	7	8	9	10	11	12
L/S on C_1	v_3	v_4				*copy*				*copy*			
ALU on C_1							v_6	v_7	v_9	v_{10}			v_{11}
L/S on C_2	v_1	v_2				*copy*	*copy*			*copy*			
ALU on C_2							v_5		v_8				v_{12}

(a) A schedule by our scheduling heuristic

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L/S on C_1	v_1	v_2						*copy*	*copy*			*copy*				
ALU on C_1							v_5				v_8					
L/S on C_2	v_3	v_4						*copy*								
ALU on C_2							v_6	v_7			v_9	v_{10}			v_{12}	v_{11}

(b) A schedule by UAS heuristic

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13
L/S on C_1	v_3	v_4				*copy*		*copy*	*copy*					
ALU on C_1							v_6	v_7						
L/S on C_2	v_1	v_2												
ALU on C_2							v_5		v_8		v_9	v_{10}	v_{11}	v_{12}

(c) A schedule by Integrated heuristic

Table 3: Schedules by different heuristics

reduce the execution time of the basic block by 3 cycles and 1 cycle, respectively .

The UAS heuristic [3] schedules the ready instructions cycle by cycle and does not reconsider earlier cycles that are already scheduled. Therefore, the *copy* instruction has to be issued at the current cycle. For example, in Table 3b instruction v_8 could be scheduled at cycle 8 if the *copy* instruction could be inserted at cycle 5 rather than cycle 7. This makes the execution time longer as the communication delay can not be hidden by other instructions.

The Integrated heuristic [4] does not consider the resource constraints when prioritising the instructions. We can see in Table 2 the instruction mobility and number of successors are not as accurate as our $l_{max}(v_i)$-successor-tree-consistent deadlines. Furthermore, the cost function for cluster assignment in the Integrated heuristic can not schedule the instructions as early as possible comparing to our instruction scheduling heuristic. For example, in Table 3c the instruction v_9 has two predecessor v_1 and v_6 on cluster 2 and cluster 1, respectively. According to their cost function, v_9 is scheduled on cluster 2 at cycle 10 as v_9's successor v_{12} has the other predecessor v_8 scheduled on cluster 2. According to our heuristic, v_9 is scheduled at cycle 8 on cluster 1.

After the $l_{max}(v_i)$-successor-tree-consistent deadline of each instruction v_i is computed, our instruction scheduling heuristic takes $O(n \log n)$ time to construct a schedule for the basic block by sorting all the instructions in non-decreasing order of their deadlines. Therefore, the time complexity of our entire instruction scheduling heuristic is dominated by the algorithm for all the $l_{max}(v_i)$-successor-tree-consistent deadlines. As a result, the following Theorem holds.

THEOREM 2. *Given a basic block, the time complexity of our entire instruction scheduling heuristic is $O(ne)$, where n is the number of instructions in the basic block, and e is the number of edges in the precedence-latency graph.*

4. SIMULATION RESULTS

To evaluate the performance of our scheduling heuristic, we simulated our scheduling heuristic, UAS heuristic [3] and Integrated heuristic [4] by using 808 basic blocks selected from MediaBench II benchmark suite [18]. The number and size range of all the basic blocks selected from each of the benchmarks and the names of the benchmarks from which each basic block is taken, are shown in Table 4. Most of the basic blocks of the benchmark suite are quite small. We selected large basic blocks from 20 to 379 instructions. In order to generate large basic blocks, we unrolled loops from the benchmark suite to obtain basic blocks with at least 1280 instructions.

Our target clustered VLIW processors have two types of functional units: load/store unit and ALU unit. The instruction set of our target processors is the instruction set of TMS320C6X processor [19], where the latencies of the load instruction, the $16 * 16$ multiply instruction and the branch instruction are 4, 1 and 5 cycles, respectively. The latencies of other instructions are 0 cycles. In addition, we assume the inter-cluster communication latency is 2 cycles. The register files are assumed to have unlimited size. The following six different clustered VLIW processors are used for the evaluation:

1. Processor (a) has 4 clusters with 2 ALU units and 2 load/store units in each cluster.
2. Processor (b) has 3 clusters with 2 ALU units and 2 load/store units in each cluster.
3. Processor (c) has 2 clusters with 2 ALU units and 2 load/store units in each cluster.
4. Processor (d) has 4 clusters with 1 ALU unit and 1 load/store unit in each cluster.
5. Processor (e) has 3 clusters with 1 ALU unit and 1 load/store unit in each cluster.
6. Processor (f) has 2 clusters with 1 ALU unit and 1 load/store unit in each cluster.

Fig. 3 shows the simulation results for the real basic blocks taken from MediaBench II benchmark suite, and Fig. 5 shows the simulation results for the artificial basic blocks unrolled from the selected loops taken from MediaBench II benchmark suite. The simulation results shown in Fig. (i) were obtained by using Processor (i). For example, Fig. 3(a) assumes Processor (a). For UAS heuristic, we used ALAP and

Benchmark	MPEG2	JPEG2000	H.264	H.263	MPEG4	JPEG
Basic Block Number	82	76	273	73	212	92
Smallest Basic Block Size	20	20	20	20	20	20
Biggest Basic Block Size	116	120	379	163	211	108
Average Basic Block Size	38	30	42	39	50	37

Table 4: Information of benchmark

CWP priority schemes which yield the best performance for UAS heuristic [3]. In each figure, a vertical axis represents the average number of processor cycles of a schedule for basic blocks from a benchmark on a specific clustered VLIW processor.

The improvements of our heuristic over UAS heuristic and Integrated heuristic are shown in Fig. 4 and Fig. 6. As we can see, our scheduling heuristic outperforms UAS heuristic and Integrated heuristic for all the selected basic blocks. On average, for the six processor models, our heuristic improves 25%, 25%, 33%, 23%, 26%, 27% over UAS heuristic, respectively, and 15%, 16%, 15%, 9%, 20%, 8% over Integrated heuristic, respectively.

There are several reasons that our heuristic performs better than UAS heuristic and Integrated heuristic. Firstly, our priority scheme considers the processor resource constraints while the priority schemes of UAS heuristic and Integrated heuristic ignore the processor resource constraints. As a result, our priority scheme is more accurate, especially when the instruction has many successors competing for the limited resources. Secondly, UAS heuristic can not insert the inter-cluster communication into earlier cycles while our heuristic can issue the inter-cluster communication earlier to hide the communication delay. Thirdly, the communication cost function of Integrated heuristic aims to minimise the inter-cluster communication while our heuristic aims to minimise the total execution time.

From the simulation results, we have the following key observations:

- To a large extent, the improvements of our heuristic over UAS heuristic and Integrated heuristic increase with the decreasing number of clusters, or the decreasing number of functional units of each cluster. The reason is that when the contention for functional units (resource constraints) among all ready instructions becomes less intense, the $l_{max}(v_i)$-successor-tree-consistent deadline of each instruction v_i starts to degenerate to the priority derived from the priority scheme that does not consider the resource constraints.

- Both UAS heuristic and Integrated heuristic suffer from a performance anomaly: the performance degrades when the number of clusters increases. Our heuristic does not have this performance anomaly. For example, when the instructions in Fig. 1 are scheduled on Processor (b), the total execution time is increased by 2 cycles comparing to that on Processor (c) for both UAS heuristic in Table 5b and Integrated heuristic in Table 5c. The reason is as follows. UAS heuristic and Integrated heuristic schedule ready instructions as early as possible, which may schedule two predecessors of an instruction on different clusters. In the case where instructions do not have enough parallelism comparing to the clusters, if the ready instructions are scheduled as early as possible, it may introduce unnecessary inter-cluster communication delay. Consider Example 1 and Table 5. If we increase the number of clusters by one, UAS heuristic and Integrated heuristic will schedule both v_3 and v_4 at the same cycle on different clusters, resulting in an extra delay of 2 cycles for v_6.
  Comparing to UAS heuristic and Integrated heuristic, our instruction scheduling heuristic takes into account the successor's execution time rather than only the current instruction's execution time. For example, in Table 5a, our heuristic tries to schedule the successor v_6 as early as possible, therefore it keeps v_3 and v_4 on the same cluster and do not utilise cluster C_3 to avoid the performance anomaly.

5. CONCLUSION

We have proposed an efficient priority-based heuristic for scheduling instructions in a basic block on a clustered VLIW processor. Our heuristic converts the original scheduling problem into a problem of scheduling the same set of instructions with a common deadline, computes the $l_{max}(v_i)$-successor-tree-consistent deadline for each instruction v_i, and uses these deadlines as priorities to construct a schedule. Compared to the previous priority schemes, our priority scheme is the first one that considers both the precedence-latency constraints and resource constraints. The priorities computed by our heuristic are more accurate with respect to the relative importance of each instruction. We have simulated our heuristic, UAS heuristic and Integrated heuristic on the 808 basic blocks taken from the MediaBench II benchmark suite using six processor models. On average, for the six processor models, our heuristic improves 25%, 25%, 33%, 23%, 26%, 27% over UAS heuristic, respectively, and 15%, 16%, 15%, 9%, 20%, 8% over Integrated heuristic, respectively.

In the existing approaches, register allocation and instruction scheduling are separated. For clustered VLIW processors, these two problems are closely related and have a significant impact on each other. An open problem is how to integrate register allocation and instruction scheduling into a single problem and solve it efficiently.

6. REFERENCES

[1] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Elsevier, 114-120, fourth edition, 2006.

[2] Andrei Terechko, Erwan Le Thenaff, Manish Garg, Jos van Eijndhoven, and Henk Corporaal. Inter-cluster communication models for clustered vliw processors. In *proceedings of Symposium on High Performance Computer Architectures*, 2003.

[3] E. Ozer, S. Banerjia, and T. M. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In

Time	0	1	2	3	4	5	6	7	8	9	10	11	12
L/S on C_1	v_3	v_4				*copy*				*copy*			
ALU on C_1							v_6	v_7	v_9	v_{10}			v_{11}
L/S on C_2	v_1	v_2				*copy*	*copy*			*copy*			
ALU on C_2							v_5		v_8				v_{12}
L/S on C_3													
ALU on C_3													

(a) A schedule by our scheduling heuristic on Processor (b)

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
L/S on C_1	v_3					*copy*		*copy*										
ALU on C_1									v_6				v_9				v_{12}	
L/S on C_2	v_4					*copy*												
ALU on C_2									v_7					v_{10}				v_{11}
L/S on C_3	v_1	v_2								*copy*	*copy*			*copy*	*copy*			
ALU on C_3							v_5				v_8							

(b) Performance anomaly by UAS heuristic on Processor (b)

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L/S on C_1	v_3					*copy*	*copy*									
ALU on C_1																
L/S on C_2	v_4									*copy*	*copy*					
ALU on C_2									v_6	v_7						
L/S on C_3	v_1	v_2														
ALU on C_3							v_5			v_8			v_9	v_{10}	v_{11}	v_{12}

(c) Performance anomaly by Integrated heuristic on Processor (b)

Table 5: Performance anomaly

Proceedings of the 31st Annual International Symposium on Microarchitecture, 1998.

[4] Rahul Nagpal and Y. N. Srikant. pragmatic integrated scheduling for clustered vliw architectures. *software-practice and experience*, 38:227–257, 2008.

[5] Jeffrey D. Ullman. *Complexity of Sequencing Problems*. John Wiley and Sons, 1976.

[6] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. The MIT Press, 1986.

[7] Saurabh Jang, Steve Carr, Philip Sweany, and Darla Kuras. A code generation framework for vliw architectures with partitioned register banks. In *proceedings of 3rd International Conference on Massively Parallel Computing Systems*, 1998.

[8] Victor S. Lapinskii and Margarida F. Jacome. cluster assignment for high-performace embedded vliw processors. *ACM transactions on design automation of electronic systems*, 7(3):430–454, July 2002.

[9] Rainer Leupers. Instruction scheduling for clustered vliw dsps. In *proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2000.

[10] Kailas K, Agrawala A, and Ebcioglu K. Cars: A new code generation framework for clustered ilp processors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001.

[11] Jesús Sánchez and Antonio Gonzálezor. Instruction scheduling for clustered vliw architectures. In *Proceedings of 13th International Symposium on System Synthesis*, 2000.

[12] Javier Zalamea, Josep Llosa, Eduard Ayguade, and Matoe Valero. Modulo scheduling with integrated register spilling for clustered vliw architectures. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 160–169, 2001.

[13] Phillip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1986.

[14] Jesús Sánchez and Antonio González. Modulo scheduling for a fully-distributed clustered vliw architecture. In *Proceedings of the 33rd International Symposium on Microarchitecture*, 2000.

[15] Josep M. Codina, Jesús Sánchez, and Antonio González. A unified modulo scheduling and register allocation technique for clustered processors. In *Proceedings of 2001 International Conference on Parallel Architecture and Compilation Techniques*, 2001.

[16] Yi Qian, Steve Carr, and Philip Sweany. optimizing loop performance for clustered vliw architectures. In *Proceedings of 2002 International Conference on Parallel Architecture and Compilation Techniques*, 2002.

[17] Alex Aleta, Josep M. Codina, Jesús Sánchez, Antonio González, and David Kaeli. Agamos: A graph-based approach to modulo scheduling for clustered microarchitectures. *IEEE Transactions on Computers*, 58(6):770–783, 2009.

[18] Mediabench ii benchmark. *http://euler.slu.edu/ fritts/mediabench/*.

[19] Ti tms320c64xx dsps. *http://www.ti.com*.

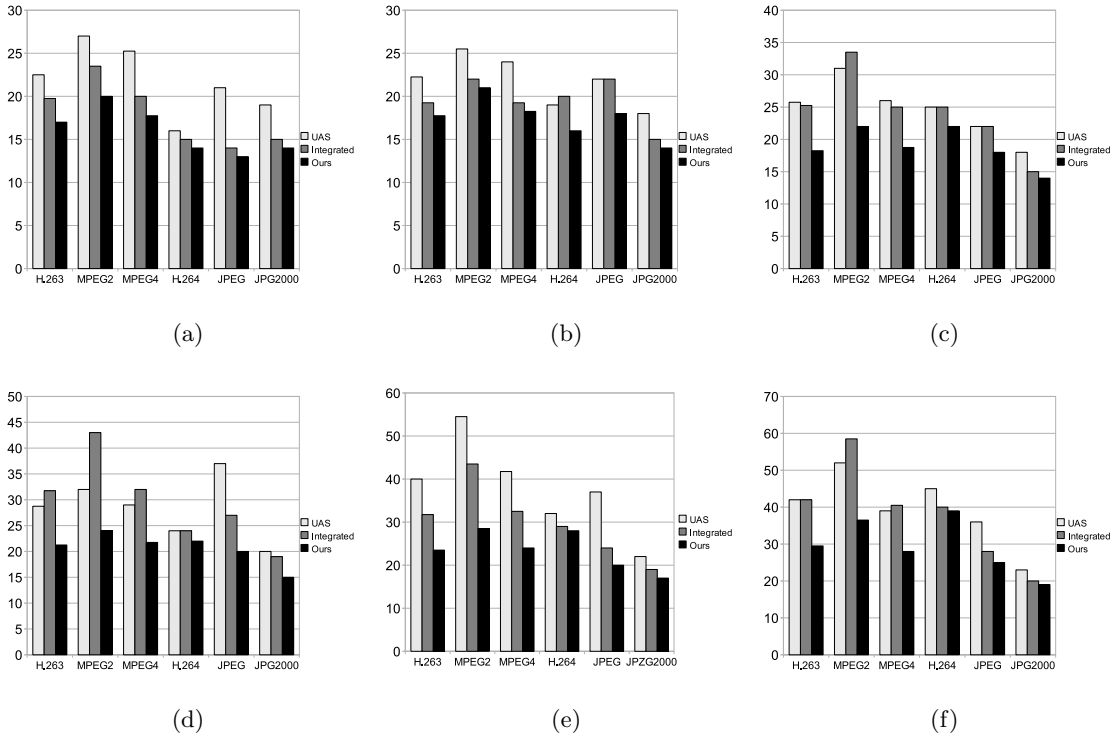

Figure 3: Simulation results Part A: Processor cycles for real basic blocks

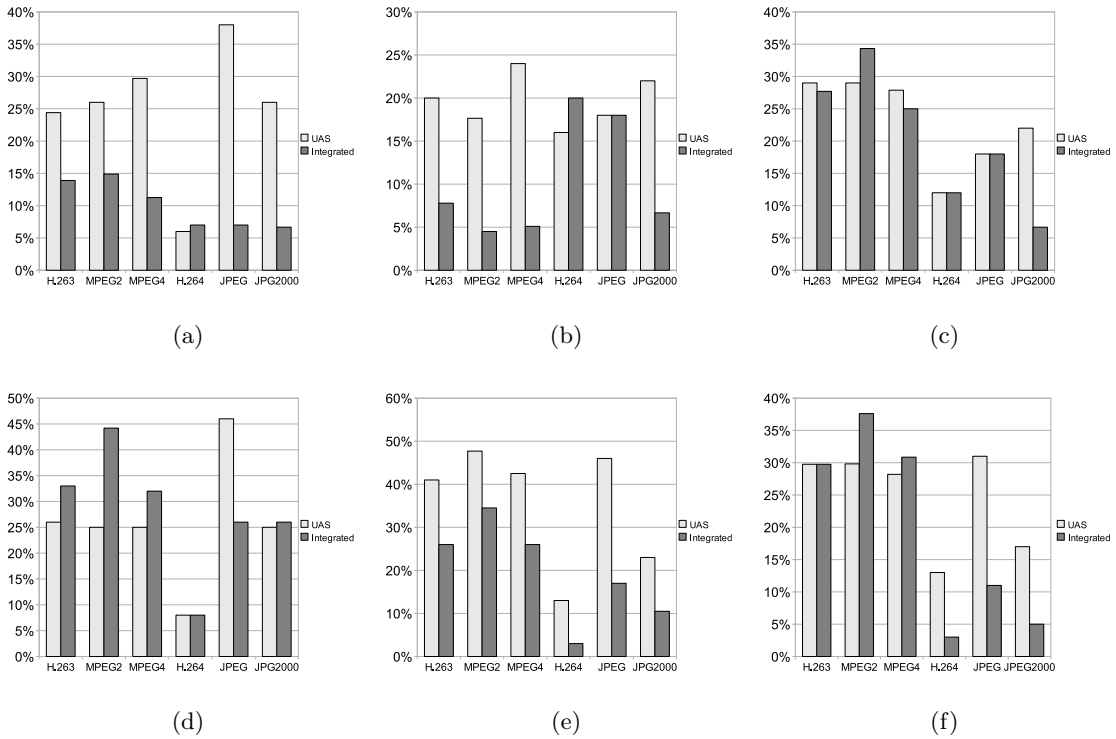

Figure 4: Simulation results Part B: Improvements of our heuristic on real basic blocks

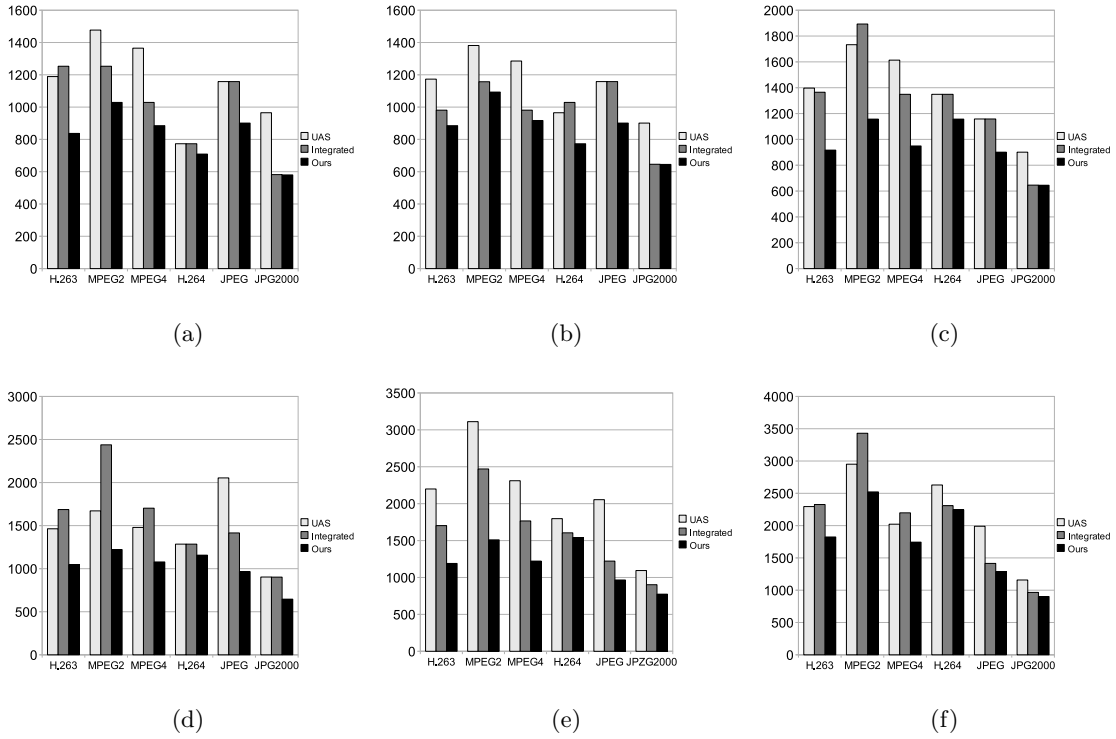

Figure 5: Simulation results Part C: Processor cycles for artificial basic blocks

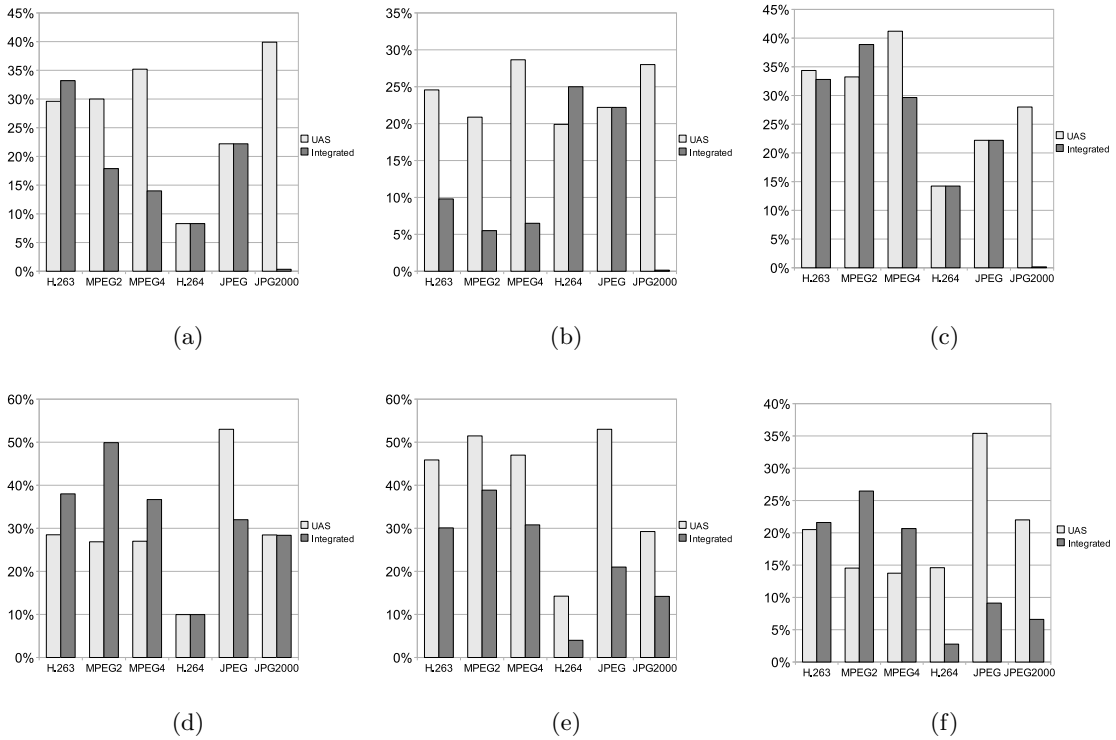

Figure 6: Simulation results Part D: Improvements of our heuristic on artificial basic blocks