Fine-Grain Real-Time Code Scheduling for VLIW Architecture

Tai M. Chung and Dae J. Hwang

Abstract

In safety critical hard real-time systems, a timing fault may yield catastrophic results. In order to eliminate the timing faults from the fast responsive real-time control systems, it is necessary to schedule a code based on high precision timing analysis. Further, the schedulability enhancement by having multiple processors is of wide spread interest. However, although an instruction level parallel processing is quite effective to improve the schedulability of such a system, none of the real-time applications employ instruction level parallel scheduling techniques because most of the real-time scheduling models have not been designed for fine-grain execution.

In this paper, we present a timing constraint model specifying high precision timing constraints, and a practical approach for constructing static schedules for a VLIW execution model. The new model and analysis can guarantee timing accuracy to within a single machine clock cycle.

I. Introduction

For a real-time system to function correctly, the controlling subsystem must be logically correct and free of timing faults. A hard real-time system can fail catastrophically if even a single operation is performed at the wrong time. Thus, it is critical that the programming system be able to ensure that all timing constraints will be met no matter what events occur at runtime.

In order to meet timing constraints, control operations are scheduled at either runtime or compile time, called dynamic or static scheduling respectively. A dynamic system determines the execution order at runtime; thus, it provides flexibility in that the system can adjust its schedule for unpredicted events, but runtime overhead limits the precision of operation timing. This overhead limits dynamic scheduling to relatively coarse grain tasks, and the imprecision of dynamic scheduling for a safety critical hard real-time system is potentially dangerous[3]. In safety critical systems, because high precision operation timing is generally necessary, the flexibility of dynamic scheduling should be sacrificed for safety[1]. The fundamental motivation of our research is that hard real-time systems require absolute freedom from timing faults.

While using a static schedule ensures safety, finding a static schedule for a sequential machine is not always feasible or even possible[5][6]. Thus, in this paper, we focus on the general problem of finding a static schedule for parallel computer architecture, more specifically for VLIW(Very Long Instruction Word) computer architectures because its synchronous behavior provides predictable execution. Each VLIW instruction called *word* contains multiple op-codes, one for each processing element within the system[8][9], yielding a very static form of parallel execution to obtain speed-up by efficiently using ILP(Instruction Level Parallelism). However, this static parallel structure can greatly increase schedulability of control programs for real-time systems -- if static analysis and scheduling of the real-time code is done at the instruction level.

Thus, this paper focuses on how the language and compiler technology can exploit instruction-level, VLIW-style, parallelism to provide valid schedules for VLIW architecture. The result is a static scheduling mechanism that takes advantage of the improved schedulability that parallelism yields, while the mechanism also provides computational speed-up in much the same way that VLIW systems were originally intended to function. In fact, improved schedulability is achieved even for coarse-grain tasks scheduled in this way, because use

Manuscript received July 28, 1995; accepted October 24, 1995.

The authors are with Department of Information Engineering, Sung Kyun Kwan University, Suwon, Korea.

of ILP tends to reduce the processing time up to the execution time of the critical path[9].

The VLIW-like scheduling techniques presented in this paper are adapted from two well known algorithms: the postpass scheme to reorganize a code for a pipeline machine[11], and the earliest deadline first algorithm for the real-time tasks[13]. Those ideas are further developed and modified to solve the real-time scheduling problem by using multiple processors in a VLIW-like system configuration.

In section 2, we describe a graphical representation of the time constrained computational model, which represents a real-time program with high-precision timing constraints. In section 3, the parallel execution model, code scheduling algorithm, and performance observation are described. In section 4, the summary and future direction of this research is presented.

II. Timing Constraint Model

A graphical representation of a real-time program is denoted as a Directed Timed Graph(DTG) G that consists of a set of nodes Γ and a set of edges Θ, i.e., $G = (\Gamma, \Theta)$. G represents a scheduling problem to find a partially ordered set of instructions that meets all timing constraints as well as precedence constraints as specified in a source program.

Let $\Gamma = \{\gamma_1, \gamma_2, \gamma_3, \cdots \gamma_n\}$ be a set of nodes to be scheduled where n is the number of nodes in G. A node is associated with a single instruction or a group of instructions as a schedulable unit. Further, each instruction associated with a node is classified as an *externally viewed instruction*(EVI) or an *internally viewed instruction*(IVI) based on the effect of the instruction. The effect of an IVI is limited to the internal computation, while an EVI may change the status of the control environment.

For example, the variables defined as *volatile* in the C language [17] or commands to control robots using RCCL [12] are EVIs, and their execution must meet the timing constraints specified in the source program. Because EVIs may depend on values computed by IVIs, any such dependence also implies a relative timing constraint or an execution order which must be preserved.

Let $\Theta = \{\Theta_k \mid k \in N \text{ and } 1 \le k \le m\}$ be a set of directed edges where m is the number of edges in G and N is a set of positive integers. An edge is associated with a timing constraint that specifies the required timing relationship between two nodes. Each directed edge in Θ is a tuple of four attributes: source node γ_b, sink node γ_e, relational operator η, and timing offset δ, i.e., a timing constraint θ_k is defined as $\theta_k = \langle \gamma_b, \gamma_e, \eta, \delta \rangle$. In particular, a timing constraint with 0 offset ($\delta = 0$) is defined as a simple constraint.

In this model, the default relation of independent constraints is a conjunction, but an *ORing* constraint can be explicitly expressed for disjunctive relations of multiple constraints participating in the constraint. An *ORing* constraint implies that meeting any one of the constraints in *ORing* constraint is sufficient. An *ORing* constraint is legal only when source nodes and sink nodes of the constraints composing the constraint are matched. Thus, one edge in G corresponds to an *ORing* constraint that may consist of multiple timing constraints.

The type of required timing relationship represented by each edge is specified by η and belongs to one of following four categories: *before*, *after*, *concurrent*, or *exclusive* constraint[1] Indeed, any temporal relations between γ_b and γ_e can be expressed by one of these constraints when it is allowed to specify *ORing* constraints. The following statements illustrate the four types of timing constraints.

- *before* constraint : $\gamma_e < \gamma_b + \delta$ (γ_e must happen no later than δ after γ_b)
- *after* constraint : $\gamma_e > \gamma_b + \delta$ (γ_e must happen no sooner than δ after γ_b)
- *concurrent* constraint : $\gamma_e = \gamma_b + \delta$ (γ_e must happen exactly at δ after γ_b)
- *exclusive* constraint : $\gamma_e \neq \gamma_b + \delta$ (γ_e must NOT happen at δ after γ_b)

The conversion of precedence or timing constraints into one of these types makes our scheduling analysis much simpler by encoding the ordering constraints in terms of the timing relationships, or by taking into account only these forms of timing constraints. A set of precedence constraints is a subset of simple constraints; thus, it can be represented without introducing any additional type of constraint. In other words, each precedence constraint of the form "γ_x *uses* γ_y*'s result*" can be encoded as the *after* constraint "γ_x *happens no sooner than 0 time unit after* γ_y".

Even though a DTG is very similar to a traditional DAG(Directed Acyclic Graph), it is different in that the edges in a DTG do not necessarily indicate precedence relations of the source and sink instructions when δ is not zero. The direction of an edge merely distinguishes the end-points(source and sink nodes) of a timing constraint between them. For example, $\gamma_e \xrightarrow{<\delta} \gamma_b$ is

1) For serial machines, concurrent constraints are unsatisfiable and exclusive constraints are trivially met.

equivalent to $\gamma_e < \gamma_b + \delta$, implying that γ_b can be executed before γ_e.

Because the number of timing constraints determines the complexity required to solve the scheduling problem, it is interesting to find the upper bound of the number of timing constraints in G. The upper bound of the number of timing constraints is a function of the number of EVIs, and fortunately less than 5% of the instructions in a real-time program are EVIs in practice.

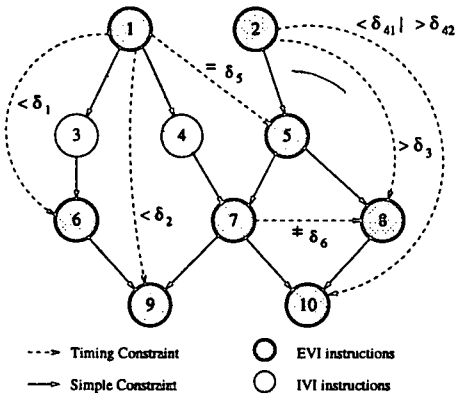


Fig. 1. Graphical Representation of a Real-Time Program.

Figure 1 depicts an example of the graphical representation of a program. Suppose the timing offset δ_i is a component of θ_i. Let us distinguish the simple constraints from timing constraints with non-zero offset for convenience. The program contains eight EVI's ($\gamma_1, \gamma_2, \gamma_5, \gamma_6, \gamma_7, \gamma_8, \gamma_9$, and γ_{10}), two IVI's (γ_3 and γ_4), six timing constraints ($\theta_1, \theta_2, \theta_3, \theta_4, \theta_5$, and θ_6), and eleven simple constraints. Among the timing constraints, two of them (θ_1 and θ_2) are *before* constraints and one (θ_3) is an *after* constraint. Another constraint (θ_4) is an *ORing* constraint that allows all the range of time except the one between δ_{42} and δ_{41}. The remaining constraints (θ_5 and θ_6) specify *concurrent* and *exclusive* constraints respectively.

III. Scheduling Algorithms

1. Parallel execution model

VLIW-like does not mean VLIW. The lack of commercial VLIW microcontrollers is only a minor inconvenience to producing a parallel control computer with the appropriate static timing constraints. As shown in Figure 2, an appropriate target system can be constructed by simply augmenting n conventional microcontrollers with barrier synchronization hardware[7]. Given such hardware, barrier synchronizations can be used to ensure that all the microcontrollers maintain lock-step execution indistinguishable from that obtained with a true VLIW architecture. The hardware barrier mechanism we use in this model is generally used for synchronization among processors[7] and more precise timing analysis[14] to minimize the number of synchronization points. Likewise, a small, statically scheduled, shared memory or communication network can provide the same functionality as the shared register file of an ideal VLIW. It is also simple for this connection to be used to allow all the microcontrollers equal and essentially independent access to the controlled devices.

The execution model supports a predictable execution time for each operation. The operation time may not be a fixed time, but expressed as a range such that an execution time, τ_{op}, could be a random value between the minimum time denoted as $L(\tau(op))$, and the maximum time denoted as $H(\tau(op))$. That is, $\tau_{op} = [\, L(\tau(op)), H(\tau(op))]$ where $L(\tau(op)) \le \tau_{op} \le H(\tau(op))$. If a controlling subsystem provides a wider range of the operation times, the difficulty of finding a valid schedule increases. For example, let γ_{i+1} be executed right after $\gamma_1, \gamma_2, \cdots, \gamma_i$. It should be guaranteed that γ_{i+1} is placed either before $\sum_{j=1}^{i}(L(\tau(\gamma_j)))$ or after $\sum_{j=1}^{i}(H(\tau(\gamma_j)))$. However, it is not guaranteed that γ_{i+1} is not placed between those times, said unclear ringe. The unclear range that γ_{i+1} can not be scheduled on is:

$$\sum_{j=1}^{i}(H(\tau(\gamma_j)) - L(\tau(\gamma_j)))$$

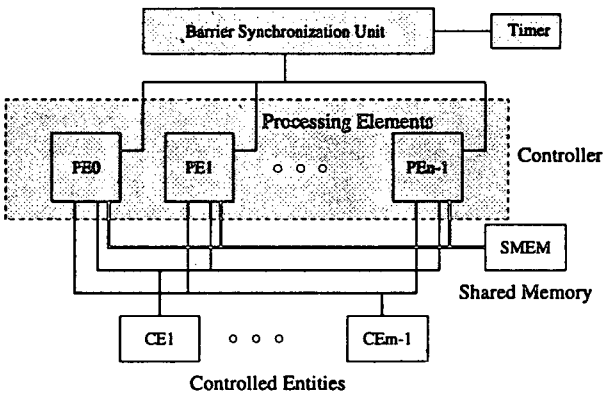


Fig. 2. An abstract parallel execution model for real-time control.

In effect, each barrier resets accumulated range to [0,0] in our timing analysis because the instructions following a barrier synchronization start simultaneously, yielding a

relative timing difference of zero. Thus, by inserting synchronization between every instructions, a VLIW execution model can be simulated where the operation time of a word equals to the maximum operation time of all the instructions in the word.

A timed delay operation is supported as one of the operations, say *delay t* implies delay for t time units. The delay operation lets the system wait any number of ticks needed. It is particularly used to enforce meeting *after* constraints by prolonging the execution time of an instruction sequence. For example, two consecutive instructions, γ_b and γ_e with timing constraint $\gamma_e > \gamma_b + \delta$ can be satisfied by inserting delay operation with delta between γ_b and γ_e. In fact, two implementaion choices can be suggested: NOPs simulating delay operation, or a barrier mechanism in cooperation with a timer (see Figure 2). If NOPs are used to simulate a delay operation, the granularity of the delay operation is the same as the NOPs granularity. Another important assumption is that the real-time system has no externally generated interrupts, although interrupts can be accepted when timing analysis is not affected (e.g., while a PE waits for a barrier or executes a time delay operation). If an interrupt without predictable interrupt latency is allowed, it is impossible to guarantee timely behavior the system.

2. *Instruction Allocation Algorithm*

Even though not being realistic to have an unlimited number of processors, it is theoretically worth to consider the case. Thus, we first develop an instruction allocation algorithm for unlimited number of processors (IAAUP), and then extend it for a given number of processors. Before proceeding, we define a few notations frequently used in this paper.

Definition:1 *Instruction slot* w_i^k *is defined as the* i_{th} *time slice reserved to execute an instruction on* PE_k(k_{th} *Processing Element). The length of* w_i^k, *expressed as* $\tau(w_i^k)$, *is the time allotted to the slice; thus,* $\tau(w_i^k) \geq \tau(\gamma)$ *should hold to allocate* γ *in* w_i^k.

Definition:2 *Word* $W_i = \{ w_i^k \mid k = 1, 2, \cdots, \ell \}$ *is a set of instructions that are allocated on the* i_{th} *time slice of every processor, and the word size* $\ell(W_i)$ *is defined as the number of instructions in* W_i[2]. *Also,* $\tau(W_i) = MAX_{j=1}^{\ell(W_i)} \tau(w_i^j)$ *denotes the length of* W_i *that is defined as the maximum length of the instruction slots in* W_i.

Definition:3 *Ready set* $\mathcal{R}$ *is a set of instructions that are ready to be allocated. The instructions in* $\mathcal{R}$ *should meet the following conditions:*

- *not yet allocated to any instruction slot*
- *not dependent on any of the unallocated instructions.*
- *not violating any* after *constraints if it is allocated in the next available instruction slot.*

The status of an instruction is classified as one of the following three types: **allocated instruction** which is already mapped into one of the instruction slots, **ready instruction** which can be allocated without violating the ready set constraint described in Definition 3, and **immature instruction** which is not ready to be allocated yet. An immature instruction either has an ancestor which is not allocated or needs more time to run so that it can meet an *after* constraint.

3. *IAA for Unlimited Number of Processors*

One of the most important properties of this environment is that all instructions in the ready set can be consumed at a time. This property makes it possible for the scheduler to place the instructions in the ready set in the same word. The description of the IAAUP is given in the Figure 3:

Inputs: $I_{in} = \gamma_1, \gamma_2, \cdots \gamma_n$ and $\Theta = \{\theta_1, \theta_2, \cdots, \theta_m\}$

Output: $A_{out} = W_1 W_2 \cdots W_k$ and $k = MAX(\ell(W_i))$
or $A_{out} = NIL$ and $k = -1$ if fails.

Procedure:

Step 1: Set variables $\mapsto$
$i = 1, R_1 = \emptyset$, and $A = \emptyset$

Step 2: Compute R_i

Step 3: $\forall \gamma_x \in R_i$, allocate γ_x to W_i
by considering the following conditions
Cond 3.1: $(\gamma_x \in Re_i) \wedge (\gamma_y \notin Re_i) \wedge ((\gamma_x = \gamma_y) \in \Theta) \mapsto$neither
Cond 3.2: $(\gamma_x \in Re_i) \wedge \gamma_y \in Re_i) \wedge ((\gamma_x = \gamma_y) \in \Theta) \mapsto$both
Cond 3.3: $(\gamma_x \in Re_i) \wedge (\gamma_y \in Re_i) \wedge ((\gamma_x \neq \gamma_y) \in \Theta) \mapsto$either one.

Step 4: If all instructions are allocated into A,
fill with NOPs and terminate

Step 5: $i = i + 1$

Step 6: Go to Step 2

Fig. 3. Algorithm description of IAAUP.

The input parameters of IAAUP are a sequence of instructions and a set of timing constraints to be met. The output parameters are the sequence of words to be executed on a VLIW machine and the number of processors needed. Initially, all the instructions are placed in the immature set(I) and the other sets(R and A) are initialized to empty sets. In Step 3, the ready set is

2) The word size is a constant for a given VLIW machine and generally equivalent to the number of processors. If less instructions than the word size are selected for allocation, the difference is filled with NOPs..

computed to satisfy the conditions as described in Definition 3. That is, the simple constraints and *after* constraints are taken care of when the ready set is computed.

In this algorithm, satisfying the *concurrent* and *exclusive* constraints is a necessary condition for the instructions to be selected. A *concurrent* constraint uses an *all or none policy* such that if one of the instructions is allocated, then all the instructions associated with the instruction via the *concurrent* constraint are allocated in the word. For the *exclusive* constraint, an *only one policy* is applied such that if an instruction is allocated, any other instruction associated with it via *exclusive* constraints are not allocated in the same word.

The conditions in Step 3 of the algorithm are given for *concurrent* and *exclusive* constraints that are associated with a pair of instructions. The extension to these conditions for multiple instructions is straightforward. Suppose Conditions 3.1 and 3.2 are considered for extended constraints, e.g., $\gamma_1 = \gamma_2 = \cdots = \gamma_p$.

If Condition 3.1 holds, none of the instructions are allocated if exists $\gamma_i \notin R, 1 \le i \le p$. In contrast, every instruction is allocated when Condition 3.2 holds.

Condition 3.3 indicates that if we have exclusive constraints specifying $\gamma_1 \neq \gamma_2 \neq \cdots \neq \gamma_p$, only one instruction is randomly selected and allocated.

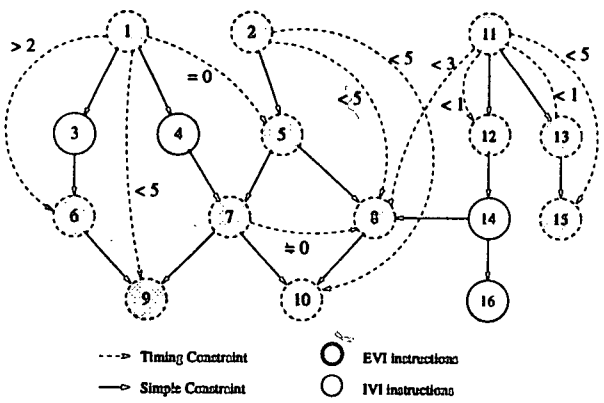


Fig. 4. A graphical representation of a code segment.

One good aspect of IAAUP is that it is optimal when the *exclusive* constraint is not considered, or optimally treated. In this paper, the term optimal algorithm implies that if there is a schedule for a code it is always found by the algorithm. In fact, the instructions scheduled by the IAAUP meet all the constraints specified in the code except *before* constraints. Also, this algorithm is optimal if Step 3 finds an instruction for the *exclusive* constraint such that the selection leads to the minimum execution time of the instructions between the source and sink instructions of the *before* constraint associated with the selected instruction.

The example shown in Figure 4 is carefully designed to show how the algorithm works with various constraints. The example contains 16 instructions: 12 EVIs and 4 IVIs, 10 timing constraints, and 17 simple constraints. Among the constraints, the DTG contains 1 *exclusive* and 1 *concurrent* constraints. Note that the graphical representation of a code has much more constraints and EVIs than real control programs in order to show the allocation process with a small example. Real control programs should have much more IVIs and fewer timing constraints.

Throughout this paper, we assume that each operation takes 1 unit of time and the communication cost is negligible to simplify the presentation. Table 1 shows how the status of each instruction is changed from immature state to allocated state.

Table 1. Instruction status changes in IAAUP.

time	immature set	ready set	allocated set
t_0	3,4,5,6,7,8,9,10,12,13,14,15,16	1,2,11	
t_1	3,4,6,7,8,9,10,13,14,15,16	1,5,12,13	2,11
t_2	6,7,9,10	3,4,8,14,15	1,2,5,11,12,13
t_3	6,9,10	7,16	1,2,3,4,5,8,11,12,13,14,15
t_4	9	6,10	1,2,3,4,5,7,8,11,12,13,14,15,16
t_5		9	1,2,3,4,5,6,7,8,10,11,12,13,14,15,16
t_6			1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

At t_1, instruction 1 is ready but cannot be allocated because of the *concurrent* constraint (*inst*1 = *inst*5). Thus, only instructions 2 and 11 are allocated at t_1. Instruction 5 which will run concurrently with instruction 1 becomes ready at t_2, and both instructions are allocated at t_2. In this example, every instruction in the immature set becomes ready immediately, when all the parent instructions are allocated, except instruction 6. Instruction 6 should be ready at t_3 if we consider only simple constraints, but the *after* constraint (*inst*6 > inst1 + 2) is not yet met. Thus, instruction 6 moves to ready state at t_5 instead of t_3. The exclusive constraint between instructions 7 and 10 (*inst*7 ≠ *inst*10) is met without any adjustment because they are naturally scheduled to run at different times (t_3 and t_5). The rest of the scheduling is straightforward, and the resulting VLIW schedule by IAAUP is shown in Table 2.

This algorithm raises an interesting question: *what is the minimum number of processors that should be allocated in this algorithm?* Let p be the necessary number of processors to execute the schedule. If m is smaller than the available processors, say k, the code can be trivially scheduled. However, it is not the usual case,

and reducing p to k or fewer processors is necessary. One approach to reduce the required number of processors is tuning the schedule to reorganize the instructions in the words so that the required number of processors becomes less than k. However, tuning is another NP-complete problem, and the timing properties built into the schedule can be destroyed. An alternative approach we take is explained in the following section.

Table 2. VLIW words scheduled from the example of code by IAAUP.

VLIW words	PE_0	PE_1	PE_2	PE_3	PE_4
Word 1	2	11	nop	nop	nop
Word 2	1	5	12	13	nop
Word 3	3	4	8	14	15
Word 4	7	16	nop	nop	nop
Word 5	6	10	nop	nop	nop
Word 6	9	nop	nop	nop	nop

1) Limited number of processors

As described in the previous section, unlimited number of processors make the scheduling problem simpler, but having unlimited number of processors is impractical. Thus, extending the IAAUP for limited number of processors is essential. This section presents the details of the heuristic algorithm we developed for k processors. First of all, a few more notations used in the following sections are defined.

Definition:4 *An open constraint $\hat{\theta}$ is defined as a constraint whose source instruction $\gamma_s(\theta)$ has been allocated, but the sink instruction $\gamma_e(\theta)$ is not allocated yet. Also, $\hat{\Theta}$ is defined for a set of open constraints at any instance. Formally,*

$$\hat{\Theta}_t = \{\theta \mid (\gamma_s(\theta) \in A \text{ at } t) \wedge (\gamma_e(\theta) \notin A \text{ at } t) \wedge (\theta \in \Theta)\}$$

Definition:5 *It is said that γ_y is reachable from γ_x, expressed as $\gamma_x \xrightarrow{*} \gamma_y$ if there exists a dependence path $d_{x,y}$, i.e., $\gamma_x \xrightarrow{*} \gamma_y$ iff $\exists d_{x,y}$.*

Definition:6 *$D_{x,y} = \{d^1_{x,y}, d^2_{x,y}, \cdots, d^m_{x,y}\}$ is a set of dependence paths from γ_x to γ_y where dependence path $d_{x,y}$ is defined as an ordered set of instructions on the directed edges with zero offset (simple timing constraints) from γ_x to γ_y in a DTG. The length of $d_{x,y}$ is defined as the total execution time of the instructions in $d_{x,y}$ or zero if $\gamma_x \xrightarrow{*} \gamma_y$. Formally,*

$$\tau(d_{x,y}) = \begin{cases} \sum_{\gamma_i \in d_{x,y}} \tau(\gamma_i) & \text{if } \gamma_i \xrightarrow{*} \gamma_j \\ 0 & \text{otherwise} \end{cases}$$

Definition:7 *An emergency path $\rho_{x,y}$ is the $d^k_{x,y}$ such that*

$$\tau(\rho^k_{x,y}) \geq \tau(d^l_{x,y}), \forall d^l_{x,y} \in D_{x,y} \text{ and } \rho_{x,y} \in D_{x,y}.$$

The emergency path of an instruction with *ORing* constraint expressed as $\theta_x = \theta^1_x \mid \theta^2_x \mid \cdots \mid \theta^m_x$ takes on the minimum length of the emergency paths from the instruction. The length of the emergency path is either the number of instructions of the path or the execution time of the path depending on the scheduling policies (see the residual based IAALP in this section). The length of the emergency path is monotonically reduced as *ORing* constraints are comprised of more subconstraints. The emergency path $\rho_{x,y}$ is expressed as:

$$\rho_x = MIN(\rho^1_x, \rho^2_x, \cdots, \rho^m_x)$$

Constraint reduction: After a constraint is open by executing an end-point of a constraint, the offset to the open constraint diminishes as time passes. Let δ_m be the original offset of θ_m, and δ^i_m be the updated offset before W_i is executed. The updated offset of the constraint after W_i is executed can be expressed as:

$$\delta^{i+1}_m = \delta^i_m - \tau(W_i)$$

If δ_m is reduced to less than zero and $\hat{\theta}$ is a *before* constraint, then the algorithm fails to schedule the code.

A zero value of δ_m implies that the instruction on $d_{s,e}$ needs to be allocated immediately to meet the *before* constraint. If δ_m is reduced to less than zero and $\hat{\theta}$ is an *after* constraint, the associated instruction can be allocated any time after δ^i becomes zero.

The constraint reduction first seems very complicated, but with a careful analysis, the complexity is reduced to a trivial problem because In our model, constraint reduction needs to be applied only to the constraints that are opened by source instructions.

Computing urgency: In order to compute the urgency of an instruction in terms of the open constraints, the length of the emergency path between the instruction and the sink instruction should be computed. The longest distance between one instruction to another is computed by Dijkstra's shortest(or longest) path algorithm with timing complexity $O(n^2)$ where the number of instructions are n. Because there can be a maximum of m open constraints at a time, the distances from each instruction to its sink instruction can be computed in $O(m \times n^2)$ time. Then, the urgency of an instruction is defined as follows where $\omega_{x,y}$ is the distance between γ_x and γ_y.

Definition:8 *ξ^t_x, urgency of an instruction γ_x at time t is defined as the number of instructions that need to be*

executed in unit time to meet all the open constraints without considering the after *constraints. Thus,*

$$\xi_x' = \begin{cases} MAX_{\forall\, \theta_j \in \Theta'}(\tau(d_{x,e(\theta_j)})/\ \hat{\delta}_j') & \text{if } \hat{\Theta}' \neq \emptyset \\ 0 & \textit{otherwise} \end{cases}$$

where $\hat{\delta}_j'$ *is the offset of the reduced constraint* $\hat{\theta}_j$, *and* $\gamma_e(\theta_j)$ *is the sink instruction of* θ_j.

Selecting instructions: In IAALP, a major extension to IAAUP is finding k instructions from R. Among the ways of selecting the instructions, the most obvious way is to select a random set of k instructions. This method is the simplest, but the processors are not utilized as efficiently as other methods that take into account timing criteria such as urgency of each instruction, implying that the schedulability is not as much improved as it could be.

The method we use in this section is to select k instructions whose urgencies are the greatest. For this decision, it is assumed that the penalty of violating each constraint is identical, and the tie conditions are resolved by a random selection.

As employed in IAAUP, the instructions associated with *concurrent* or *exclusive* constraints are evaluated according to their timing requirement. Being different from the IAAUP, the instructions associated with a *concurrent* constraint raise their urgencies to that of the one with the highest urgency among them while the instructions associated with an *exclusive* constraint ignore all the instructions, but the one with highest urgency.

Allocating instructions : When k or fewer instructions are selected, they are allocated in the instruction slots of the next word. In this case, allocating those instructions causes several side-effects. At first, the instructions in the immature set are probed, and the instructions satisfying the ready set requirement are moved into R. The instructions which have delayed for *after* constraints and the descendants of allocated instructions are the candidates to be moved. Secondly, if a source instruction is allocated, the constraint associated with the instruction is registered as an open constraint based on the constraint reduction analysis. Thirdly, if a sink instruction is allocated, the constraint associated with the instruction is withdrawn from the open constraint set.

If less than k instructions are selected, the rest of the instruction slots are filled with the appropriate number of delay operations. In this paper, we assume that each delay operation is a NOP operation and consumes one unit of time.

Description of IAALP: As shown in Figure 5, this algorithm is very similar to IAAUP except that a heuristic algorithm is used to select k instructions based on urgencies of the instructions in R.

Inputs: $s_{in} = \gamma_1, \gamma_2, \cdots \gamma_n$, $C = c_1, c_2, \cdots, c_m$,
k = Number of processors

Output: Valid schedule: $s_{out} = W_1 W_2 \cdots W_p$ or NIL if fails.

Procedure:

Step 1: Set variables $\mapsto$
$i = 1$ and $R_1 = \emptyset$
Compute $d_{x,e(\theta_j)}, \forall \gamma_x \in \Gamma, \theta_j \in \Theta$

Step 2: Compute R_i

Step 3: $\forall \gamma_x \in R_i$, Ψ = best $\hat{k} \leq k$ instructions
by considering the following conditions.
Cond 3.1: $(\gamma_x \in Re_i) \wedge (\gamma_y \notin R_i) \wedge ((\gamma_x = \gamma_y) \in \Theta) \mapsto$ neither
Cond 3.2: $(\gamma_x \in Re_i) \wedge \gamma_y \in Re_i) \wedge ((\gamma_x = \gamma_y) \in \Theta) \mapsto$ both
Cond 3.3: $(\gamma_x \in Re_i) \wedge (\gamma_y \in Re_i) \wedge ((\gamma_x \neq \gamma_y) \in \Theta) \mapsto$ one with higher $\hat{\xi}$.

Step 4: Do constraint reduction & allocate $\gamma \in \Psi$ into W_i.

Step 5: Fill nop's if $\hat{k} < k$.

Step 6: Terminate if all instructions are allocated

Step 7: $i = i + 1$

Step 8: Go to Step 2

Fig. 5. Instruction Allocation Algorithm for Limited Processors.

With the given input parameters, --a set of instructions, a set of timing constraints, and the number of available processors -- the initial computation is done before allocating instructions. The important process in this step is computing the distances of emergency paths from each instruction to the sink instructions of the constraint. These distances are used when the urgencies are computed in Step 2.

The extension to Condition 3.1 for multiple instructions is identical to the one for IAAUP. For Condition 3.2, the urgencies of all instructions associated with the *concurrent* constraints are set to the one with the highest urgency. For example, say $\xi_1' \leq \xi_2' \leq \cdots \leq \xi_p'$. The urgencies of the instructions are set to ξ_p'. Thus, the instructions are allocated in an *all or none* fashion if enough processors are available. If there is fewer than p processors available, then there are two alternatives: one is to increase the urgencies of the instructions to the maximum and the other is not to allocate any of the instructions. For Condition 3.3, if we have a constraint specifying $\gamma_1 \neq \gamma_2 \neq \cdots \neq \gamma_p$, only one instruction with the highest urgency is allocated. For example, when $\xi_p' \leq \xi_i' 1 \leq i \leq p$, only γ_p is allocated.

Selecting instructions based on residual: The previous algorithm is based on the urgency which is computed from the distance of(number of instructions on) the emergency path and the remaining time of the open constraints. This scheme is not useful when the execution times of the instructions are very different because each

instruction Is equally weighted to compute the urgency. For a system with widely different execution time of the operations, using the estimated residual time is much more effective. Residual time is the time left after all the instructions on the critical path are executed. It is defined as:

Definition:9 *The residual of an instruction* $\Delta_{x,y}$ *is defined as the expected time left when* $d_{x,y}$ *is executed, i.e.,*

$$\Delta_{x,y} = \delta_{x,y} - \sum_{\gamma_i \in d_{x,y}} \tau(\gamma_i)$$

In this modification, IAALP selects the instructions with minimum residual values. Since this scheme uses the exact execution time, it provides more accurate analysis than the urgency based selection scheme. A negative residual value of an instruction implies that the schedule already failed to meet the timing constraint if it is a *before* constraint.

2) An example of IAALP with 3 processors

Consider an example given in Figure 4 for IAALP. The process of IAALP is shown in Table 3 in chronological order. Each entry of the table has two elements. The first element is the status of the instruction indicated as I for Immature state, R for ready state, and A for Allocated state. The second element is the urgency which is the number of instructions to run within unit time to meet all the before constraints. For example, the entry for instruction 3 denotes that it has to run at least 3 instructions for 4 units of time at t_3 in order to meet the constraint $(\gamma_1 \xrightarrow{<5} \gamma_9) \wedge (\gamma_2 \xrightarrow{<5} \gamma_{10})$. Note that the urgency is valid only for the ready instructions. The suffix to the urgency indicates either *concurrency* constraint by C or *exclusive* constraint by *E*. The instructions 1 and 5 are concurrent instructions; hence, instruction 1 is not allocated until t_2 when instruction 5 becomes ready.

The allocation of the instructions in IAALP is determined by the urgency as mentioned fefore. At t_3, instruction 14 $(\xi^3_{14}=1.0)$ is selected over instruction 15 $(\xi^3_{15}=0.75)$. In fact, if the urgency is the same as the operation time, the instruction has to be allocated to meet the constraints, i.e., a necessary condition to generate a valid schedule is that the maximum value of urgency should be 1 or less in our paradigm because we assume that each operation takes one unit of time.

As shown in Table 3, when an instruction is about to open a constraints, the urgency of the instruction is set to zero, meaning that it is not urgent at all. The urgency, however, can be changed to meet other constraints associated with the instruction. The urgency of instruction 1 at t_2 is set to the urgency of instruction 5 $(\xi^2_5=0.75)$ because instruction 1 needs to run concurrently with instruction 5. In this example, we allocate concurrent instructions first by raising their urgencies, if there are more instructions with the same urgency than the number of available processors. For the instructions associated with exclusive constraints, only one instruction is selected based on the urgencies of the instructions Re. If there are more than one such instruction, a random selection is made. In the example, instructions 7 and 8 are not in Re at the same time; thus, it does not have any impact.

Table 3. Status of the instruction in IAALP.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7
inst1	R, 0.0C	R, 0.75C	R, 1.0C	A, -	A, -	A, -	A, -
inst2	R, 0.00	A, -	A, -	A, -	A, -	A, -	A, -
inst3	I, -	I, -	I, -	R, 0.75	A, -	A, -	A, -
inst4	I, -	I, -	I, -	R, 0.75	A, -	A, -	A, -
inst5	I, -	R, 0.75C	R, 1.0C	A, -	A, -	A, -	A, -
inst6	I, -	I, -	I, -	I, -	I, -	R, 0.66	A, -
inst7	I, -	I, -	I, -	I, -	A, -	A, -	A, -
inst8	I, -	I, -	I, -	R, 0.66E	A, -	A, -	A, -
inst9	I, -	I, -	I, -	I, -	I, -	I, -	R, 0.00
inst10	I, -	I, -	I, -	I, -	I, -	R, 1.00	A, -
inst11	R, 0.00	A, -	A, -	A, -	A, -	A, -	A, -
inst12	I, -	R, 1.00	A, -	A, -	A, -	A, -	A, -
inst13	I, -	R, 1.00	A, -	A, -	A, -	A, -	A, -
inst14	I, -	I, -	R, 1.00	A, -	A, -	A, -	A, -
inst15	I, -	I, -	R, 0.25	R, 0.33	R, 0.5	A, -	A, -
inst16	I, -	I, -	I, -	R, 0.0	R, 0.00	A, -	A, -

Table 4. Scheduled words for 3 processors for the example.

VLIW words	PE_0	PE_1	PE_2
Word 1	2	11	nop
Word 2	12	13	nop
Word 3	1	5	14
Word 4	3	4	8
Word 5	7	15	16
Word 6	6	10	nop
Word 6	9	nop	nop

Note that the emergency distances of γ_{11} over the remaining time to γ_{13} and γ_{15} are 1.0 and 0.2 for the constraints $\gamma_{13} < \gamma_{11}+1$ and $\gamma_{15} < \gamma_{11}+5$ respectively. In this case, the maximum value of 1.0 is taken as the urgency of γ_{11} as stated in Definition 8.

The selection does not guarantee to lead to an optimal solution, meaning that a valid schedule may not be found although it exists. The quality of IAALP depends on the strategy to select instructions in Re; thus, by replacing the

selecting module, we can improve the quality of the algorithm. We learned that the IAALP performs well for reasonably complicated problems as discussed in the next section. The resulting VLIW schedule generated by IAALP is shown in Table 4.

4. Performance Analysis and Discussion

The scheduling time of IAA is very small; hence, it is capable of handling large problems. Consider the IAALP procedure in Figure 5. Step 1 requires $O(m\times n^2)$ time because Dijkstra's shortest path algorithm requires $O(n^2)$ time and it is applied for every constraint and instruction pair. In Step 2, computing the ready set costs a constant amount of time. In step 3, the instructions in Re are probed for each condition. Because the size of Re is n in the worst case, the time complexity of Step 3 is $O(n^2)$. The constraint reduction in Step 4 costs $O(n\times m)$ time in the worst case when each instruction is associated with its constraints. Thus, the computational complexity of the IAALP is:

$$T = O(m\times n^2)+O(n^2)+O(n^2)+O(n\times m)$$
$$= O(m\times n^2)$$

IAALP has a saturation number of processors for a problem, meaning that the execution time is not reduced when the number of processors used exceeds a certain number. By increasing processors beyond this saturation point, the processor utilization is degraded without improving the performance at all. The processor utilization ratio π is defined as:

$$\pi = \frac{\text{number of slots used}}{\text{total number of slots}}$$
$$= \frac{\text{number of slots used}}{\text{number of processors} \times \text{number of words}}$$

If the execution time is not reduced, IAA does not improve schedulability. Table 5 shows the execution times(number of words) and the utilization ratio when different number of processors are used. For example, when 4 PEs are used to run 80 instruction code(32 words), the processor utilization ratio is computed as:

$$\pi = \frac{80}{4\times 32}$$
$$= 0.625$$

In the experiment, we see that the saturation point becomes larger when there exists more parallelism in the program. The degree of parallelism is determined how much of dependencies are enforced. In Table 5, the sequences of 64 and 128 instructions have larger saturation points(5 processors) that other sequences(4 processors) because they have more parallelism than other s[3)]

Table 5. Execution times of a schedule for k processors.

Code size	2 PE_s	4 *PEs*	6 *PEs*	8 *PEs*
64	32	16	13	13
80	43	32	32	32
96	51	37	37	37
112	62	46	46	46
128	68	48	46	46

Figure 6 depicts the processor utilization ratio π based on the number of processors used. π is very slowly reduced when i is less than the saturation point. Then, it is reduced very fast, and eventually it would be reduced to zero, i.e. $\pi_\infty = 0$, which is the processor utilization ratio when an infinite number of processors are used.

While developing IAA, we have learned that there is a challenging problem: *how are the instructions mapped on multiple processors?* This problem is simple when communication cost is insignificant(e.g., shared register system, pure VLIW) because the processor-instruction mapping does not change any timing property. If the communication cost is identical for every processor(e.g., the shared memory system), the instructions mapped into any processor have the same timing properties unless some of them are in registers. Only the instructions which access data in remote PE have different timing property.

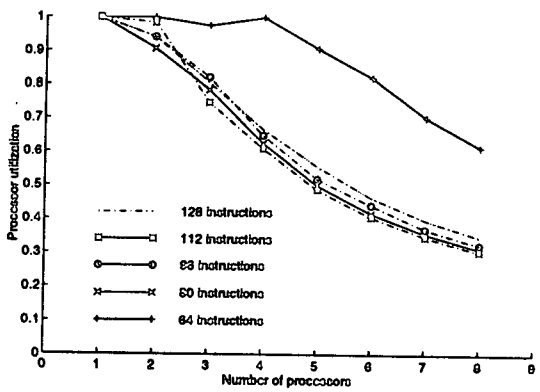


Fig. 6. Processor utilization when k processors are used.

However, the most complex problem is the processor-instruction mapping when communication cost is

3) we carefully built our example to have almost equivalent degree of parallelism except sequences of 64 and 128 instructions that are modified to have more parallelism by eliminating some simple constraints.

not identical (e.g., distributed memory system). The timing analysis for this case is beyond scope of this paper, but very actively studied by many research groups [2][4][15][16].

IV. Conclusion and Future Research

In this paper, we design a time-constrained model that allows the specification of high precision timing constraints. Unlike most real-time system models, our model is designed for fine-grain static timing analysis. Such a timing analysis provides accurate and robust runtime behavior by eliminating dynamic scheduling and dispatching overhead and by scheduling the code at instruction level.

VLIW-like scheduling algorithm to convert graphical representation of the model into a sequence of words is developed based on the code reorganization algorithm for pipeline architecture and the EDF policy for real-time scheduling This approach is straightforward, but the performance and complexity of the algorithm show that it is usable for either a pure VLIW architecture or a MIMD architecture simulating a VLIW execution model. As we have seen in Section 3.4, this algorithm can be also used to schedule thousands of instructions within reasonable time. For larger problems, hierarchical decomposition methodology has been proposed in [5]. Currently, IAALP algorithm is implemented in CHaRTS as a module of a real-time compiler. In CHaRTS, an intermediate representation of a real-time code is reorganized to generate words for VLIW machine. It shows, as discussed in Section 3.4, that schedulability is improved by employing more processors. Also, further development is being made to apply IAALP algorithm for more general execution model, assuming asymmetric communication cost, distributed memory access, layered memory structure, or heterogeneous architecture need to be explored. Also, IAALP algorithm is employed in CHaRTS as a module of a real-time compiler.

References

[1] Putting a stop to vehicles slipping and sliding. *Control Systems*, page 24, July 1994.

[2] J.W. Baugh and W.M. Elseaidy. Timing analysis of a multiprocessor architecture for active control. In *Proceedings of the 11th Conference on Analysis and Computation*, pages 203 -- 212, Atlanta, GA, April 1994.

[3] Scheduling Hard Real-Time Systems: A Review. *Software Engineering Journal*, 6(3):116 -- 128, May 1991.

[4] J.Y. Choi, I.Lee, and I.Kang. Timing analysis of superscalar processor programs using acsr. In *Proc. of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, pages 63 -- 67, Seattle, WA, May 1994.

[5] T.M. Chung. *CHaRTS: Compiler for Hard Real-Time Systems*. PhD thesis, Purdue Unversity, West Lafayette, IN, Aug 1995.

[6] T.M. Chung and H.G. Dietz. Language constructs and transformation for hard real-time programs with fine-grained timing constraints. In *Proc. of ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, pages 45 -- 53, La Jolla, CA, June 1995.

[7] W.E. Cohen, H. G. Dietz, and J.~B. Sponaugle. Dynamic barrier architecture for multi-mode fine-grain parallelism using conventional processors. In *Proc. of International Conference of Parallel Processing*, volume II, St. Charles, IL, August 1994.

[8] J.R. Ellis. *Bulldog : a compiler for VLIW architectures*. MIT Press, Cambridge, MA, 1986.

[9] E.B. Fernandez and B. Bussell. Bounds on the number of processors and time for multiprocessor optimal schedule. *IEEE Transactions on Computers*, c-22(8):745 -- 751, August 1973.

[10] J.A. Fisher. New architecture for supercomputing. In *32nd IEEE Computer Society International Conference*, pages 177 -- 180, San Francisco, CA, Feb 1987.

[11] T.Gross. Code optimization of pipeline constraints. Technical Report TR-83-255, Stanford University, December 1983.

[12] J.S. Lee, S. Hayati, V.Hayward, and J. E. Lloyd. Implementation of RCCL, a Robot Control C Library on a MicroVAX II. In *Intelligent Robots and Computer Vision*, pages 472 -- 480, Cambridge, MA, Oct 1988.

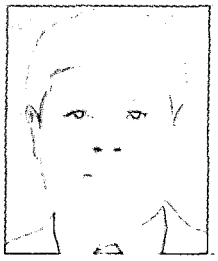
[13] C.L. Liu and J.Layland. Scheduling algorithms for multiprocessing in a hard real-time environments. *Journal of ACM*, 20(1):46 -- 61, January 1973.

[14] M.T. O'Keefe and H. G. Dietz. Barrier MIMD Architecture: Design and Compilation. Technical Report TR-EE 90-50, Purdue University, West Lafayette, Indiana, August 1990.

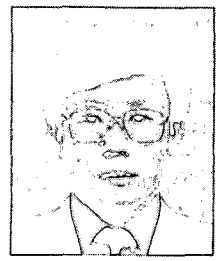
[15] A.Shaw. Deterministic timing schema for parallel programs. *IEEE Computers*, 24(5):56 -- 63, May 1991.

[16] H.F. Wedde, B. Korel, and D.M. Huizinga. Formal timing analysis for distributed real-time programs. *Real-time Systems*, 7(1):57 -- 90, July 1994.

[17] ANSI X3.159. *Programming Language C*. American National Standard Institute, 1989.



Tai Myoung Chung received his B.S.(Electrical Engineering) degree from Yonsei University, Korea in 1981, B.S.(Computer Science) and M.S. (Computer Engineering) degrees from University of Illinois, Chicago, U.S.A. in 1984 and 1987 respectively, and Ph.D. degree from Purdue University, Indiana, U.S.A. in 1995. Between 1985 and 1990, he worked at Waldner and Co. and at Bolt Beranek and Newman Labs. where he involved in Automated Network Management project. He is currently an assistant professor of Information Engineering department at Sung Kyun Kwan University in Korea. His research interest includes real-time systems, distributed systems, parallel compiler and architecture. He is a member of IEEE and ACM.



Dae Joon Hwang received his M.S. and Ph.D. degrees in computer science from Seoul National University, Seoul, Korea in 1981 and 1986, respectively. He has been a professor of Information Engineering Department at Sung Kyun Kwan University, Korea, since 1987. His research has been much focused on multithreaded computer architecture design, parallel processing system, and load balancing. Now, he is developing multimedia application crafting workbench. DooRae in which he is trying to incoorperate the idea from parallel processing into real-time multimedia collaboration. Before joining the SKKU, he was an assistant and associate professor of the Department of Computer Science at Han Nam University, Taejon, Korea from 1981 to 1987. From 1990 he spent a year working with Prof. Arvind at MIT. Also, he conducted research on multithreaded computer architecture with Dr. K. Ekanadham, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, between November 1993 and March 1994. He is a member of the KISS, the AACE, the ACM and the IEEE and its several SIG's.