Conditional Scheduling for Embedded Systems Using Genetic List Scheduling

Martin Grajcar
grajcar@lrs.uni-passau.de
University of Passau, Chair of Computer Architectures

Abstract

One important part of a HW/SW codesign system is the scheduler which is needed in order to determine if a given HW/SW partitioning is suitable for a given application. In this paper, we employ a dataflow model for scheduling a computation including conditional branches *on a loosely coupled heterogeneous multiprocessor system. The goal is to minimize the worst-case makespan while satisfying constraints implied by data dependencies and exclusive resource usage.*

We present a formal model which allows multiple schedule optimizations and a new efficient heuristic approach based on genetic algorithms and list scheduling.

1. Introduction

There are a lot of interesting scheduling problems in many areas of computer science ranging from High-Level Synthesis to Hardware-Software Codesign. The solution of a scheduling problem (the *schedule*) consists of two parts:

- Assigning tasks and communications to resources
- Determining the start times of tasks and communications

This paper concentrates on one important feature: *conditional execution* of tasks. Because of this, these assignments and start times may depend on results of previously computed tasks. Moreover, only a subset of the task set need to be executed (see sect. 1.2).

A formal problem description will be given in sect. 2. Because of scheduling problems being NP-complete (except if extremely simplified [4]), the need for a good heuristic is obvious. There are many different scheduling algorithms (see sect. 3).

Our approach is an extension to a combination of *genetic algorithms* (GA) and *list scheduling* (see sect. 4).

According to our experimental results (given in sect. 5), this algorithm works for conditional scheduling nearly as well as the basic algorithm described in [5] does for the non-conditional scheduling problem.

1.1. Basic scheduling problem

Our scheduling problem is built on top of a simple (albeit NP-complete) scheduling problem of mapping a *task graph* onto a target architecture (taken from [5]). The limited space does not allow discussing of other problem extensions.

The task graph is an acyclic directed graph (T, C). The node set (T) contains *tasks* and the edge set (C) contains *communications* (all non-preemptable).

The target architecture consists of a set of (processing) *modules* M and a set of *busses* B. Each bus is assumed to be connected to some of the modules (usually there is only one bus connected to all modules). Each module consists of a CPU (responsible for the computation), a local memory, and a communication processor for each connected bus, so computation and communication can overlap. The CPUs may be general purpose processors, DSPs, and/or ASICs.

We define the *resource set* $\boldsymbol{R} = M \cup B$ and the *user set* $\boldsymbol{U} = T \cup C$. This way it is possible to treat all users (and all resources) uniformly, thus significantly simplifying the problem formulation (and implementation).

We define the *resource graph* $(\boldsymbol{R}, \boldsymbol{W})$ where $\boldsymbol{W}$ denotes the *wiring set*. The resource pair (r, r') belongs to $\boldsymbol{W}$ iff data can be transported directly from r to r'. For example if module m can write onto bus b, then $(m, b) \in \boldsymbol{W}$.

Using the *user graph* $(\boldsymbol{U}, \boldsymbol{D})$ with the *dependency set* $\boldsymbol{D} = \{(t, (t', t'')) \in T \times C : t = t'\} \cup \{((t, t'), t'') \in C \times T : t' = t''\}$, tasks and communications are treated uniformly.

There is a function d defined on $\boldsymbol{U} \times \boldsymbol{R}$ returning the *execution time* $d_{u,r} \in [0, \infty]$ of user u on resource r ($d_{u,r} = \infty$ denotes an illegal assignment). There are four cases:

For $(u, r) \in T \times M$ the value of $d_{u,r}$ is simply the time it takes to compute task u using module r. For $(u, r) \in C \times B$ it is the time it takes to transfer the data of u over bus r. For $(u, r) \in T \times B$ we set $d_{u,r} = \infty$ as it is not possible to compute a task using a bus. For $(u, r) \in C \times M$ we set $d_{u,r} = 0$ as an execution of a communication on a module does not correspond to any data transfer. Such a *internal* execution is only allowed if both the source and the destination of u are also executed on r [5]. In sect. 2.3 a more general requirement will be given obsoleting the one above.

We denote the *start time*, the *end time* and the *assignment* ass_u of user u by σ_u, η_u, and ass_u, respectively ($\eta_u = \sigma_u + d_{u,\mathsf{ass}_u}$. During the whole *execution interval* $[\sigma_u, \eta_u)$ the resource r may not be used by any other user.

A user can start execution only after having received all its data, i.e., after all its predecessors (i.e., all incoming communications for a task, or the source task for a communication) have finished The goal is to find the schedule with the least *makespan* defined as $\mathsf{makespan} = \max_{u \in \boldsymbol{U}} \eta_u$, which satisfies the conditions described above.

1.2. Conditional execution

In this context an *ordinary user* can be executed, if it receives one *token* on every of its input edges. Each executed user places a token on every of its output edges.

Conditional execution are usually described using the following two special users:

- *switch-user* This is a user with two incoming edges (one control and one data). According to the value of the boolean token on the control input, the token from the data input is moved on one of the two output edges.
- *select-user* This is a user with three incoming edges (1 control and 2 data). According to the value of the boolean token on the control input, one token is moved from one of the data inputs to the only output (see fig. 1).

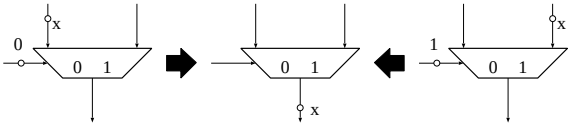

Fig.1 The special user select

Using the two special users, arbitrary branches can be modeled. A switch-user is used so that users in the wrong branch get no token and a select-user moves on the token from the appropriate branch. In the left example in fig. 2 user u_0 gets a token from u_a only if u_c produces a token with value 0 (otherwise u_1 gets the token). Anyway, the token produced by either u_0 or u_1 will be sent to u_b.

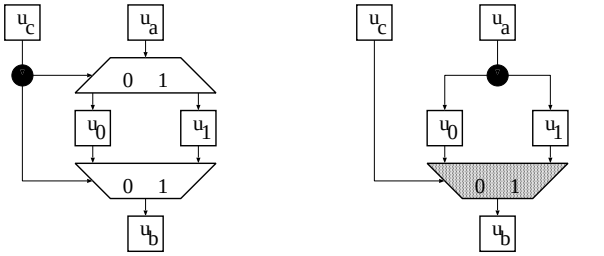

Fig.2 Two models of a branch

There are some problems with this model, e.g., it is very easy to make it behave quite strangely (e.g., by swapping the data inputs of the switch). Moreover, this model introduces unnecessary data dependencies. In fact, u_0 and u_1 do not depend on u_c and could be computed before u_c finishes.

In the context of dataflow it is assumed that users have no side-effects, so both u_0 and u_1 may be executed. This is expressed using the so-called *multiplexor* (see the right part of fig. 2), which takes tokens from both of its data inputs, moves one of them (selected by the boolean token removed from the control input) to the output, and discards the other.

We call a user connected to a multiplexor via a control edge a *control user*. Such users affect the schedule as their results determine if a given user is to be executed.

The unnecessary execution of users producing discardable results can be avoided by the scheduler. For example in fig. 2 the scheduler can decide to execute u_c first and thereafter u_0 or u_1 (thus saving unnecessary execution) or to execute all these three users concurrently (if possible).

Multiplexors are also sufficient for modeling arbitrary branches. Moreover, the behavior of the user graph is easy to understand even in complicated cases like the one in fig. 3. Note that modeling this example using switch and select is quite complex and counterintuitive.

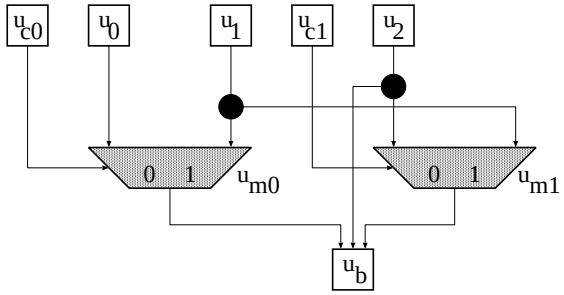

Fig.3 Sophisticated use of multiplexors

Things get complicated considering more users and more non-identical resources. With multiple multiplexors it is not even obvious how to define a schedule, so we need to introduce some formalism (see sect. 2).

1.3. Previous related research

The basic problem considered in this paper has been studied very extensively [5, 7, 1]. The conditional scheduling problem is much more complex, and there is only relatively few corresponding literature. The algorithm described in [3] starts with one schedule for each possible condition and merges them (see sect. 4.1). In [6] an algorithm is given allowing tasks in different branches to share resources. It assumes the problem to be represented in a hierarchical fashion and does not consider speculative execution.

An algorithm for high-level synthesis is given in [9] which treats scheduling of conditional branches and loops. The algorithm performs well as it employs a global analysis of suitability of different scheduling alternatives. The algorithm is based on the notion of control step which is appropriate as in the high-level synthesis all tasks usually takes one or two steps. However, tasks in our problem have execution times being arbitrary real numbers.

The algorithm from [10] is based on Ordered Binary Decision Diagrams and is able to provide all optimal schedules for control/data paths. Unfortunately, this algorithm employs control steps too and therefore cannot be applied.

2. Problem description

2.1. Conditional scheduling problem

Let $\bar{\mathbb{R}}_{\geq} = [0, +\infty]$, $\mathbb{B} = \{0, 1\}$ and let further $(A \to B)$ denote the set of all functions from A to B.

Definition: The *conditional scheduling problem* is a tuple $\mathbf{p} = (\boldsymbol{U}, \boldsymbol{D}, \boldsymbol{R}, \boldsymbol{W}, d, \boldsymbol{C}, \mathtt{app})$ satisfying the following:

- $\boldsymbol{U}$ and $\boldsymbol{R}$ are disjoint finite sets
- $(\boldsymbol{R}, \boldsymbol{W})$ is a directed graph (called the resource graph)
- $(\boldsymbol{U}, \boldsymbol{D})$ is an acyclic directed graph (called the user graph)
- $d \in ((\boldsymbol{U} \times \boldsymbol{R}) \to \bar{\mathbb{R}}_{\geq})$ is a function returning the execution time of a user on a resource
- $\boldsymbol{C} \subseteq \boldsymbol{U}$ is the set of control users

- $\mathsf{app} \in ((\boldsymbol{D} \times (\boldsymbol{C} \to \mathbb{B})) \to \mathbb{B})$ is a function describing the conditional behavior (see below)

Using example from fig. 3 and the assumptions that there are three modules m_0, m_1, m_2 connected via a single bus b and that m_0 and m_1 share memory, we state the following:

$$\begin{aligned}
\boldsymbol{R} &= \{m_0, m_1, m_2, b\} \\
\boldsymbol{W} &= \{(r,r) : r \in \boldsymbol{R}\} \cup (\{b\} \times \boldsymbol{R}) \cup (\boldsymbol{R} \times \{b\}) \cup \\
&\quad \{(m_0, m_1), (m_1, m_0)\} \\
\boldsymbol{U} &= \{u_0, u_1, u_2, u_{c0}, u_{c1}, u_{m0}, u_{m1}, u_b\} \\
\boldsymbol{D} &= (\{u_{c0}, u_0, u_1\} \times \{u_{m0}\}) \cup (\{u_1, u_{c1}, u_2\} \times \{u_{m1}\}) \\
&\quad \cup (\{u_{m0}, u_2, u_{m1}\} \times \{u_b\}) \\
\boldsymbol{C} &= \{u_{c0}, u_{c1}\}
\end{aligned}$$

For an element c of $(\boldsymbol{C} \to \mathbb{B})$ we use the term *condition*. Let $c = \{u_{c0} \mapsto x, u_{c1} \mapsto y\}$ denote a function for which $c(u_{c0}) = x$ and $c(u_{c1}) = y$ hold. Now we can write

$$(\boldsymbol{C} \to \mathbb{B}) = \big\{\{u_{c0} \mapsto 0, u_{c1} \mapsto 0\}, \{u_{c0} \mapsto 0, u_{c1} \mapsto 1\}, \{u_{c0} \mapsto 1, u_{c1} \mapsto 0\}, \{u_{c0} \mapsto 1, u_{c1} \mapsto 1\}\big\}.$$

The value of $\mathsf{app}_{(u,u'),c}$ determines if $(u,u') \in \boldsymbol{D}$ applies under $c \in (\boldsymbol{C} \to \mathbb{B})$. One data dependency applies iff u_{c0} returns 0 ($\mathsf{app}_{(u_0,u_{m0}),c} = 1 - c(u_{c0})$), one applies iff u_{c0} returns 1 ($\mathsf{app}_{(u_0,u_{m1}),c} = c(u_{c0})$), and the others apply always.

2.2. Conditional schedule

In order to define a schedule, we have to specify for every user a flag φ_u determining if it is to be executed. If so, we then need to specify its start time σ_u and assignment ass_u.

Such a specification may depend on values returned by the control tasks (so we need $\varphi_{u,c}$ instead of φ_u, etc.). This leads directly to the following

Definition: A *conditional schedule* is a tuple

$$\mathbf{s} = (\varphi_{u,c}, \sigma_{u,c}, \mathsf{ass}_{u,c})_{(u,c) \in \boldsymbol{U} \times (\boldsymbol{C} \to \mathbb{B})} \in ((\boldsymbol{U} \times (\boldsymbol{C} \to \mathbb{B})) \to (\mathbb{B} \times \overline{\mathbb{R}}_{\geq} \times \boldsymbol{R})).$$

As an example consider $c = \{u_{c0} \mapsto 0, u_{c1} \mapsto 1\}$. Now the equation $\varphi_{u,c} = 1$ means that, whenever u_{c0} returns 0 and u_{c1} returns 1, the user u is to be executed. Under the same condition $\sigma_{u,c} \in \overline{\mathbb{R}}_{\geq}$ and $\mathsf{ass}_{u,c} \in \boldsymbol{R}$ are the start time of u and its assigned resource, respectively. These values are meaningless in the case $\varphi_{u,c} = 0$ and we set $\sigma_{u,c} = \infty$.

2.3. Valid conditional schedule

One great advantage of the above definition is that the validity requirements stated for a (unconditional) schedule can be easily adapted for a conditional schedule. The disadvantage is the need for defining $3 \cdot |\boldsymbol{U}| \cdot 2^{|\boldsymbol{C}|}$ variables. As typical values for real examples are $|\boldsymbol{U}| \leq 100$ and $|\boldsymbol{C}| \leq 6$, this is not as bad as it seems to be.

Let $\eta_{u,c} = \sigma_{u,c} + d_{u,\mathsf{ass}_{u,c}}$ denote the *end time* of a user u under the condition c. Let further $\Delta_{c,c'} = \{u_c \in \boldsymbol{C} : c(u_c) \neq c'(u_c)\}$ be the set of control users on which c and c' differ.

Definition: A conditional schedule $\mathbf{s}$ for problem $\mathbf{p}$ is *valid* iff it satisfies the following conditions:

- *Data dependencies* Let c be a condition and (u,u') be a dependency applying under c. Then u' can only be executed if u is also executed. Moreover, u' cannot start before the end of u and the resources assigned to the users need to be connected in order to be able to transfer data.

$$\bigvee_{c \in (\boldsymbol{C} \to \mathbb{B})} \bigvee_{(u,u') \in \boldsymbol{D}} \big(\mathsf{app}_{(u,u'),c} \wedge \varphi_{u',c} = 1\big) \Rightarrow \big(\varphi_{u,c} = 1 \wedge \eta_{u,c} \leq \sigma_{u',c} \wedge (\mathsf{ass}_{u,c}, \mathsf{ass}_{u',c}) \in \boldsymbol{W}\big)$$

  Note that the requirement $(\mathsf{ass}_{u,c}, \mathsf{ass}_{u',c}) \in \boldsymbol{W}$ replaces the requirement on internal execution stated in sect. 1.1.
- *Exclusive resource usage* Execution intervals of two users assigned to the same resource may not overlap.

$$\bigvee_{c \in (\boldsymbol{C} \to \mathbb{B})} \bigvee_{\substack{u,u' \in \boldsymbol{U} \\ u \neq u'}} \varphi_{u,c} = 0 \vee \varphi_{u',c} = 0 \vee \mathsf{ass}_{u,c} \neq \mathsf{ass}_{u',c} \vee [\sigma_u, \eta_u) \cap [\sigma_{u'}, \eta_{u'}) = \emptyset$$

- *Causality* If a user u is executed under condition c and not under condition c', then these conditions must differ by a control user, which ends before u starts. The same holds if u is executed under both conditions, but the corresponding start times or assignments differ.

$$\bigvee_{c,c' \in (\boldsymbol{C} \to \mathbb{B})} \bigvee_{u \in \boldsymbol{U}} \bigvee_{((\varphi_{u,c}, \sigma_{u,c}, \mathsf{ass}_{u,c}) \neq (\varphi_{u,c'}, \sigma_{u,c'}, \mathsf{ass}_{u,c'}))} \bigvee_{u_c \in \Delta_{c,c'}} \varphi_{u_c,c} = 1 \wedge \eta_{u_c,c} \leq \sigma_{u,c}$$

  As the result of a control user consist of only one bit, we assume it to be instantly available on all resources.

3. Scheduling algorithms

We can here only describe algorithms important for this work, i.e., scheduling (sect. 3.1) and GA (sect. 3.2).

3.1. List scheduling

The term *list scheduling* denotes a large class of scheduling algorithms (see [7] for an excellent analysis). Here we describe a scheme which most of such algorithms comply with. This section does not consider conditional scheduling.

A list scheduler is an algorithm employing the sets:

- $R \subseteq \boldsymbol{U}$ the *remaining set* consisting of users yet to be scheduled
- $E \subseteq R$ the *eligible set* consisting of users which can be scheduled in the current step
- $F_r \subseteq \overline{\mathbb{R}}_{\geq}$ the *free set of a resource* r being a union of time intervals in which r is not in use

Initially, we set $R := \boldsymbol{U}$, $E := \{u' \in R : \nexists_{u \in \boldsymbol{U}} (u,u') \in \boldsymbol{D}\}$, and $\forall_{r \in \boldsymbol{R}}\ F_r := \overline{\mathbb{R}}_{\geq}$. Thereafter the following three steps are repeated until R is empty:

- *User selection* One user $u \in E$ is selected. Usually, the set E is implemented as a list (this is where the term "list scheduling" comes from) sorted in decreasing order of *user priorities* π_u.
- *Resource selection* A resource r is selected for user u according to some criterion. It is convenient to compute some priorities $\pi_{u,r}$ for all $r \in \boldsymbol{R}$, and to select the resource with the highest priority.
  The priority $\pi_{u,r}$ can be set equal to the negative of the earliest possible end time of user u on resource r, so the resource minimizing η_u will be selected [5].

- *Updating of the sets* User r is removed from the remaining set R and from the eligible set E. The eligible set is extended by all users having no more predecessors in the remaining set. The free set F_r of resource r is updated by removing the execution interval of user u. Formally, the following assignments are made:
  $R := R \setminus \{u\}$ $\qquad F_r := F_r \setminus [\sigma_u, \eta_u)$
  $E := \{u' \in R : \nexists_{u \in R} (u, u') \in \boldsymbol{D}\}$
  Users considered later by the scheduler can be executed earlier, if there is an appropriate interval in F_r (this feature is called "hole filling"). Of course, this way better schedules can be obtained, because the requirements on the user priority values are a little bit relaxed.
  For efficiency, the free set should be implemented as a tree of intervals. For simplicity, sometimes only the last of these intervals is maintained (which can be represented solely by its starting point). This can be expressed by the assignment $F_r := F_r \setminus [0, \eta_u)$.
  Recomputing the priorities on every iteration makes the scheduler slower but may lead to better schedules.

Existing list scheduling algorithms differ mainly by the way the priorities π_u and $\pi_{u,r}$ are defined. By computing priorities in a sophisticated manner they try to schedule the most critical users first and to assign them to the appropriate resource. The algorithms work in a deterministic fashion, and terminate after creating a schedule.

3.2. Genetic algorithms

The term *genetic algorithm* (GA) is used for an optimizing algorithm working according to the "survival of the fittest" principle [8]. It employs a set of *individuals* called *population* which is usually initialized at random. An individual somehow represents a solution to the problem. In the context of scheduling it can be a valid schedule (as defined in sect. 2.3) or a guidance for building a schedule (e.g., a vector of numbers to be used as priorities by a list scheduler).

The better an individual, the higher its so-called *fitness* (we could set $\mathsf{fitness} = -\mathsf{makespan}$). Starting with two (or more) individuals (so-called *parents*) one (or more) new individual(s) (so-called *child(ren)*) is created using so-called *genetic operators*. Better individuals are selected more often as parents while worse individuals are removed from the population with a higher probability.

There should be a fair chance that the "good features" of the parents combine in the child eventually producing an individual better than all earlier individuals. For that reason the design of a genetic operator is not easy.

The task of combining valid schedules to a new valid schedule is quite complicated [8]. The chances that the resulting schedule is better than its parents can be very low as the following argument demonstrates.

Let $\boldsymbol{R}$ be a set consisting of n identical resources and let $\mathbf{s}$ be a good valid schedule. Let further $\mathbf{s}'$ be a schedule obtained from $\mathbf{s}$ by permuting $\boldsymbol{R}$. Obviously, $\mathbf{s}'$ is valid and as good as $\mathbf{s}$. While combining the two parents $\mathbf{s}$ and $\mathbf{s}'$ common genetic operators take some assignments from the former and the remaining ones from the latter (usually choosing at random) [8].

The probability is very high that in the child schedule one resource is overused. The probability is even higher that two tasks u and u' are assigned to the same module in both $\mathbf{s}$ and $\mathbf{s}'$, but to different modules in the child schedule. This means that a communication between them can be executed internally in the parents but not in the child, possibly deteriorating the schedule.

So even in the ideal case of combining two essentially equal schedules there is little hope for producing a good schedule. Less straightforward representations (like priority vectors) perform much better [5, 2]. The explanation for this phenomenon is simple: A good individual assigns a high priority to users which have to be executed early. Combining such individuals results in a child also assigning a high priority to critical users. The problem described above does not emerge since the list scheduler takes care of proper assigning the resources to users.

Summarizing, we can say that using individuals containing an indirect representation of a problem solution (like a priority vector) rather then the solution itself (like a schedule) makes the computation of the fitness more time consuming (since a schedule must be built using a list scheduler). This disadvantage is nullified by the fact that the chances of producing good individuals are much better. Additionally, it is much easier (and faster) to combine priority vectors than to combine valid schedules.

4. Genetic list scheduling

The algorithm described here is based on the algorithm given in [5]. As we deal with a much more complicated problem, we have to omit many details not directly related to the *conditional* scheduling.

4.1. Basic ideas

We could try to create a schedule for each condition $c \in (\boldsymbol{C} \to \mathbb{B})$ and to merge such schedules in a conditional schedule like in [3], but we do not believe it to be a good idea for the following reasons:

- We would need to solve $|(\boldsymbol{C} \to \mathbb{B})| = 2^{|\boldsymbol{C}|}$ scheduling problems (NP-complete!).
- Because of the problem complexity, some heuristic has to be used. A poor solution to any of the numerous problems could deteriorate the resulting conditional schedule.
- Distributing the time among the problems is not easy.
- While solving the problems separately, the solutions can differ not only in the users which are executed and their start times (given by $\varphi_{u,c}$ and $\sigma_{u,c}$), but also in the assigned resources (given by $\mathsf{ass}_{u,c}$).
  We see no way how to reasonably merge schedules differing also in the assignments (the algorithm from [3] does not seem to be extensible this way).

Our algorithm considers the problem as a whole and uses the following principles:

- A population is maintained using a GA (see sect. 3.2).
- Each individual contains a priority vector (see sect. 4.2). Individuals are combined using genetic operators described in [5].
- Some kind of list scheduler is used for generating a conditional schedule (sect. 4.3).

4.2. Priority vectors

The most straightforward approach would be to associate a priority to each user $u \in \boldsymbol{U}$. Obviously, such priority vectors contain only few information. As only the ordering imposed by the priority vector matters (rather than the exact numbers), the list scheduler can generate at most $|\boldsymbol{U}|!$ different schedules. This number is evidently too small compared to the huge search space.

Another straightforward approach would be to associate a priority to each pair $(u,c) \in \boldsymbol{U} \times (\boldsymbol{C} \to \mathbb{B})$. This would very often lead to schedules with $(\varphi_{u,c}, \sigma_{u,c}, \mathsf{ass}_{u,c}) \neq (\varphi_{u,c'}, \sigma_{u,c'}, \mathsf{ass}_{u,c'})$ for two conditions $c, c' \in (\boldsymbol{C} \to \mathbb{B})$.

Because of the causality requirement, this would enforce the execution of some control user $u_c \in \Delta_{c,c'}$ before u. There are $2^{|\boldsymbol{C}|}$ conditions $c \in (\boldsymbol{C} \to \mathbb{B})$ and each of these may force us to schedule some of the $|\boldsymbol{C}|$ control users before u, so we are very likely forced to do that, ending up with a schedule in which all control users are executed as soon as possible This corresponds with the the switch-select model and precludes important optimization possibilities.

For this reason, we prefer associating a priority to pairs consisting of a user u and a subset h of $(\boldsymbol{C} \to \mathbb{B})$, so that u is considered by the list scheduler at the same time under all conditions $c \in h$.

We employ only subsets h called *hypercubes* of the form

$$h = h_{c,c'} = \{c'' \in (\boldsymbol{C} \to \mathbb{B}) : \Delta_{c,c''} \subseteq \Delta_{c,c'}\}$$

for two conditions $c, c' \in (\boldsymbol{C} \to \mathbb{B})$. The set $h_{c,c'}$ is in fact the smallest hypercube containing both c and c'. Let $\boldsymbol{H}$ denote the set of all $3^{|\boldsymbol{C}|}$ hypercubes $h_{c,c'}$.

So we associate a priority $\pi_{u,h}$ to each pair (u,h) and interpret it as described in the next section. In order to reduce the computational overhead, we use a sparse representation for priority vectors (assuming $\pi_{u,h} = 0$ for all pairs not mentioned).

4.3. The conditional list scheduler

Because of lack of space we can not describe the list scheduler here in full detail. Fortunately, it is not necessary as the functionality of the scheduler can be derived from the ideas given above.

We outline only the most important features of the list scheduler:

- The *remaining set* R contains pairs (u,h) instead of users. The same holds for the *eligible set* E.
- The pair $(u, h_{c,c'}) \in R$ is eligible, iff all all data dependencies applying under all conditions $c'' \in h$ are satisfied and all control users needed for determining if a condition belongs to h have been computed. Written formally:

$$E = \Big\{ (u, h_{c,c'}) \in R : \bigvee_{c'' \in h} \Big(\bigvee_{(u',u'') \in \boldsymbol{D}} (u = u'' \wedge \mathsf{app}_{u,c''}) \Rightarrow \varphi_{u,c''} = 1 \Big) \wedge \Big(\bigvee_{u_c \in \boldsymbol{C}} c(u_c) = c'(u_c) \Rightarrow \varphi_{u_c,c''} = 1 \Big) \Big\}$$

- While processing a pair (u,h) the scheduler considers scheduling u under all conditions $c \in h$ at once with the following exceptions:
  - ◇ Condition $c \in h$ under which u has already been scheduled is ignored.
  - ◇ A condition under which some predecessor of u has not yet been scheduled is also ignored. This way the data dependency requirement from sect. 2.3 is never violated.

  Considering the conditions from the whole hypercube h helps to satisfy the causality requirement from sect. 2.3.

5. Experimental results

GA combined with list scheduling have already been shown to be efficient for scheduling for embedded systems [5, 2]. Our results demonstrates that such algorithms can be extended to handle also conditional scheduling problems.

We generated for every condition $c \in (\boldsymbol{C} \to \mathbb{B})$ a *reduced user graph* G_c including only users which must be computed under c. For every such reduced user graph a (unconditional) schedule was generated using a genetic list scheduling algorithm similar to [5].

The algorithm evaluated for each reduced graph G_c 200 individuals (starting with a population of size 25). Our auxiliary experiments show that using more individuals is pointless.

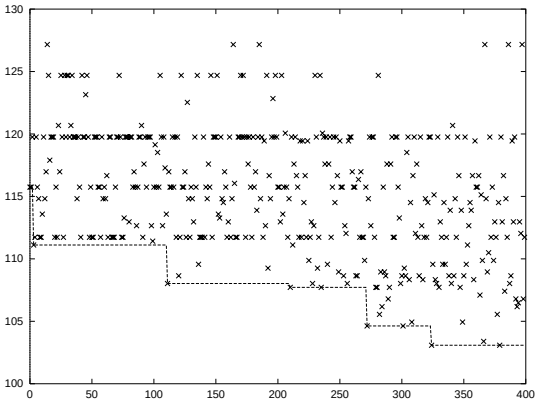

Fig.4 Typical evolution of relative schedule length depending on the number of individuals. The crosses represent individual schedules while the line represents the best individual generated so far.

Since the conditional schedule must allow for every possible condition, its makespan is not shorter than the optimal makespan for any reduced graph G_c. So we compute for each generated conditional schedule $\mathbf{s}$ the so called *relative schedule length* defined as $\frac{l}{\max_{c \in (\boldsymbol{C} \to \mathbb{B})} l_c} \cdot 100\%$ where l denotes the length of $\mathbf{s}$ and l_c denotes the length of the best schedule for G_c found.

Figure 4 shows how the relative schedule length evolves (as a function of number of individuals). We are only interested in the best schedule found so far.

We repeated the experiment for examples with 1 to 6 control users so there were up to 64 reduced user graphs to be scheduled in each experiment. We let the number of tasks take the values 32, 64, and 128, so there are $6 \cdot 3$ different parameter settings.

Let x_n denote the relative schedule length for the best of the first n individuals generated by the conditional scheduler. We report the average value of x_n over 10 runs for selected values of n in table 1. We also report the average times τ_0 and τ_1 in milliseconds the unconditional and the conditional scheduler need for the generation of one schedule, respectively.

Table 1. Relative schedule length and run times

$\|C\|$	$\|T\|$	number of individuals						times	
		25	50	100	200	300	400	τ_0	τ_1
1	32	113.5	113.2	111.9	106.5	103.6	102.2	35	40
1	64	111.5	108.6	106.4	104.9	103.5	102.5	66	77
1	128	108.3	106.8	106.0	104.7	103.8	103.1	131	156
2	32	117.7	115.3	114.4	112.3	111.6	110.8	34	67
2	64	111.3	110.6	109.3	104.5	103.4	102.4	66	131
2	128	109.4	106.7	105.5	104.1	102.2	101.0	132	271
3	32	118.4	116.6	116.3	113.4	113.4	111.9	34	122
3	64	112.2	111.5	109.9	107.7	107.6	107.3	65	240
3	128	111.3	109.8	108.8	106.4	105.4	105.0	128	491
4	32	116.8	114.4	113.6	109.8	109.2	107.6	35	245
4	64	113.4	112.8	110.9	109.9	108.6	106.9	66	460
4	128	111.9	111.1	110.2	108.1	106.9	105.8	123	909
5	32	114.9	112.4	110.1	105.7	105.4	103.7	32	441
5	64	120.2	119.0	116.4	115.6	113.9	113.2	65	928
5	128	113.5	111.5	110.5	108.9	107.6	106.9	122	1770
6	32	112.1	111.6	109.6	108.4	107.4	106.8	32	923
6	64	120.5	119.6	117.2	115.2	113.7	112.8	52	1481
6	128	112.7	112.0	110.1	108.8	107.4	106.8	113	3608

With increasing number of control users the conditional scheduler needs more time for generating one schedule. The ratio grows quickly with the number of control users since the variables $(\varphi_{u,c}, \sigma_{u,c}, \mathsf{ass}_{u,c})$ must be computed for each condition (there is some room for optimizations here).

Anyway, even for our largest example containing 6 control users and 128 tasks the run time of the scheduler (evaluating 400 individuals) does not exceed 4 seconds (recall that HW/SW codesign is a lengthy process) while producing reasonable results. Although we do not know the length of the optimum schedule, there is a strong indication, that our results lie within about 15% of the optimum:

- According to [5] and our auxiliary experiments, the genetic list scheduler performs very well for the unconditional problem and typically returns schedules lying within 5% of the optimum.
- The length of the conditional schedule exceeds the length of the unconditional one only by about 10%.

According to [3], running times for the merging 32 schedules for a graph containing 120 tasks are about $0.3s$. This time does not include the time needed for computing the schedules for all reduced user graphs G_c. We estimate this time as $2^{|C|} \cdot \tau_0$ corresponding with about $7s$ for our largest example.

An important advantage of our approach compared to the approach from [3] is the ability to benefit from speculative execution. Moreover, the algorithm from [3] requires to make the user assignments independent on the condition (thus possibly excluding many good schedules).

Unfortunately, our algorithm can not be compared to [6] as we do not assume the problem to be represented hierarchically. Other related algorithms (e.g. [9]) we are aware of solve different problems (e.g. high-level synthesis), so no comparison is possible.

6. Conclusion and outlook

A formal model for the conditional scheduling problem has been given which provides for multiple optimization possibilities. Principles of the conditional scheduler were outlined and its performance evaluated. To our knowledge, our scheduler is the only one allowing scheduling speculative execution in the context of heterogeneous multiprocessor systems.

Albeit our current implementation misses some important features of [5] (e.g., selecting from multiple genetic operators and advanced parents selection mechanism), our algorithm performs well both in terms of running speed and result quality.

We plan on removing some programming inefficiencies so the time the conditional scheduler need to generate a schedule grows only slowly with the number of control users and to incorporate the features mentioned above.

7. References

[1] A. Bender: *Design of an Optimal Loosely Coupled Heterogeneous Multiprocessor System;* in European Design&Test Conference 1996, Paris 1996, p. 275

[2] M. K. Dhodhi, I. Ahmad, R. Storer: *SHEMUS: Synthesis of heterog. multiproc. systems;* in Microproc. and Microsys., Vol. 19, No. 6, p. 311, 1995

[3] Eles, Kuchcinski, Peng, Doboli, Pop: *Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems;* Proc. DATE, Paris, 1998

[4] Garey, Johnson: *Computers and intractability - a guide to the th. of NP-completeness;* Freeman, 1979

[5] M. Grajcar: *Genetic List Scheduling Alg. for Scheduling and Allocation on a Loosely Coupled Heterog. Multiproc. System;* 36th DAC, New Orleans, 1999

[6] T. Kim, N. Yonezawa, Jane W. S. Liu, C. L. Liu: *A Scheduling Algorithm for Conditional Resource Sharing - A Hierarchical Reduction Approach;* in IEEE Trans. on IC and Systems, Vol. 13, No. 4, 1994

[7] Yu-Kwong Kwok, I. Ahmad: *Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task graphs to Multiprocessors;* in IEEE Trans. on Parallel and Distributed Systems, Vol. 7, p. 506, 1996

[8] Z. Michalewicz: *Genetic Algorithms + Data Structures = Evolution Programs;* Springer, 1996

[9] R. Moreno, R. Hermida, M. Fernamdez, H. Mecha: *A unified approach for scheduling and allocation;* Integration, the VLSI Journal 23, p. 1, 1997

[10] I. Radivojevic, F. Brewer: *Symbolic Techniques for Optimal Scheduling;* Proc. Synth. and Simulation Meeting and Int. Interchange, SISAMI, 1993