A Genetic-Algorithm-Based Reconfigurable Scheduler

David Montana, Talib Hussain and Gordon Vidaver

BBN Technologies, Cambridge MA 02138, USA
dmontana@bbn.com, thussain@bbn.com, gvidaver@bbn.com

Summary. Scheduling problems vary widely in the nature of their constraints and optimization criteria. Most scheduling algorithms make restrictive assumptions about the constraints and criteria and hence are applicable to only a limited set of scheduling problems. A reconfigurable scheduler is one that, unlike most schedulers, is easily configured to handle a wide variety of scheduling problems with different types of constraints and criteria. We have implemented a reconfigurable scheduler, called Vishnu, that handles an especially large range of scheduling problems. Vishnu is based upon a genetic algorithm that feeds task orderings to a greedy scheduler, which in turn allocates those tasks to a schedule. The scheduling logic (i.e. constraints and optimization criteria) is reconfigurable, and Vishnu includes a general and easily expandable framework for expressing this logic using hooks and formulas. The scheduler can find an optimized schedule for any problem specified in this framework. We illustrate Vishnu's flexibility and evaluate its performance using a variety of scheduling problems, including some classic ones and others from real-world scheduling projects.

1 Introduction

Most optimizing schedulers solve a limited class of scheduling problems in a single domain. In contrast, a reconfigurable scheduler can solve a wide range of different problems across a variety of domains. Using a reconfigurable scheduler, a user should be able to switch easily between scheduling, for example, taxicab pickups, athletic fields, factory machinery, classrooms, and service visits.

Although the term "reconfigurable scheduler" is new, the concept of reconfigurable scheduling has existed for decades, and a variety of reconfigurable schedulers have been created. However, the progress in this area has been slow recently. To a large extent, this reflects the view that the existing reconfigurable scheduling frameworks, such as AMPL [1] and OPL Studio [2], are as powerful as possible, and offer little room for improvement. However, we claim that our more recently developed reconfigurable scheduler, called Vishnu, is a significantly better approach that is closer to the ideal of full coverage of real-world scheduling problems.

Like other reconfigurable schedulers, Vishnu has both an optimizer and a problem representation framework, as first described in lesser detail in [3]. Unlike other

such schedulers, the optimizer is a hybrid of a genetic algorithm and a greedy schedule builder. The genetic algorithm generates orderings of the task to schedule, and the schedule builder adds one task at a time to the schedule in that order. In both the genetic algorithm and schedule builder are logic hooks where the user can specify key pieces of the scheduling logic, such as the optimization criterion, the task durations, and amount of capacity consumed. The user specifies these using a formula language that is similar to those used in modern spreadsheet programs. For each hook, the user either specifies a formula for computing the result or accepts the default formula. Vishnu also provides additional components, such as a graphical user interface and configurable statistical tables.

Section 2 provides a brief background of reconfigurable scheduling and evolutionary scheduling, with the focus of the latter on those schedulers closest to our particular approach. Section 3 gives an overview of the Vishnu optimizer and problem representation framework, with an emphasis on a high-level view of its capabilities and philosophy. Section 4 examines Vishnu in greater detail, describing how each of the different types of constraints is implemented. Section 5 demonstrates the performance and capabilities of Vishnu on both classical scheduling problems and complex real-world problems. While the former are useful for benchmarking, the latter more accurately showcase Vishnu's power to allow quick development of solutions to complex and unique scheduling problems. Section 6 concludes the paper.

2 Background

Our work is at the intersection of two distinct threads of scheduling research: reconfigurable scheduling and evolutionary scheduling. We now discuss each of these threads.

2.1 Reconfigurable Scheduling: The Concept

A reconfigurable scheduler has two main components: a problem representation framework and an optimizer (also called a solver). The problem representation framework provides a means for a user to specify the hard and soft constraints of a scheduling problem. Hard constraints cannot be violated and determine what constitutes a legal schedule. Soft constraints are preferences that need not be satisfied, but violating them causes a decrease in schedule quality; hence, they determine what constitutes a good schedule and define the optimization criterion. Generally, the framework includes a language to represent these constraints. The optimizer searches for a schedule that satisfies all the hard constraints and that optimally (or at least nearly optimally) trades off between the different soft constraints. Preferably, the optimizer can solve any problem specified in the framework. Additional components are also highly desirable. A graphical user interface allows a user to view, modify, and otherwise interact with the schedules created. A database allows multiple users to interact with the same scheduling problem. Configurable statistics gathering allows a user to create statistical tables matched to the problem.

There are some clear advantages to a reconfigurable scheduler [4]. Primarily, it greatly reduces both the time and cost of developing a scheduler for a new scheduling problem. If the nature of the scheduling problem changes, it is quick and easy to modify to incorporate these changes. A reconfigurable scheduler is reusable, so a user does not have to develop or purchase, and then learn how to use, a different scheduler for each scheduling problem. So, the existence of an effective and easy-to-use reconfigurable scheduler would make optimized scheduling available to a much larger set of users.

A reconfigurable scheduler cannot solve every scheduling problem. There will always be some type of constraint that cannot be represented in the problem representation framework. The goal for a reconfigurable scheduler is to approach the ideal, handling as many different types of real-world scheduling problems and scheduling concepts as possible. It should be capable of being easily extended to cover new concepts to ensure that it can grow towards the ideal. Furthermore, the problem representation framework should make it easy for a user to define scheduling problems, and the optimizer should perform a reasonably efficient search for a schedule.

2.2 Reconfigurable Scheduling: Previous Work

The separation of the problem representation from the schedule generation process, which is central to reconfigurable scheduling, is not a new idea. Both the mathematical programming community and constraint programming community have a history of work on problem representation languages and associated solvers.

For mathematical programming, AMPL [1] is the most popular modeling language and serves as a good representative of its class. Competitors and predecessors, such as GAMS [5], have similar functionality. AMPL allows representation of the algebraic constraints and optimization criteria used in mathematical programming. There exist multiple solvers, including CPLEX [6], that generate solutions to the problems modeled in AMPL. From the viewpoint of reconfigurable scheduling, there are two major shortcomings of the AMPL approach. First, the solvers generally cannot solve all problems representable in AMPL. For example, CPLEX can only solve problems amenable to linear programming, mixed integer programming, or convex optimization. Second, there is a limited representation capability in AMPL for logical, as opposed to algebraic, constraints. As an example, it would be hard to represent the following constraint in AMPL: "If it is later than 10:00 and a resource has already done a full hour of work executing tasks and has not rested yet, then the resource should rest for 10 minutes."

Constraint programming also has its own languages and frameworks for representing scheduling problems. Some examples are CHIP [7], Prolog III [8], and ODO [9]. The solvers associated with constraint programming are usually tree-based search algorithms. One traditional shortcoming of the constraint programming approach is an inability to express constraints involving complex algebraic expressions. A second shortcoming is that the tree search method is ineffective at global optimization. Recent developments have addressed these problems. A new constraint

programming language, OPL [2], allows representation of complex algebraic constraints in addition to logical constraints. There are associated OPL solvers, such as those available in ILOG's OPL Studio product. Improvements to the tree-based search technique, such as those described in [10], have increased the optimization performance. Still, there are a variety of types of constraints that cannot be represented in OPL, and hence many real-world scheduling problems that cannot be solved. So, there is still much room for improvement.

Reconfigurable scheduling is a much more recent development in the evolutionary computation, as well as the larger metaheuristic, community. In addition to Vishnu, there have been some other efforts at making reconfigurable genetic schedulers, including [11] and [12], but these lack the generality of Vishnu.

An area closely related to reconfigurable scheduling is scheduling ontologies [13, 14]. A scheduling ontology is a set of vocabulary, concepts and relations that can be used to describe and characterize different scheduling problems. It is essentially equivalent to the problem representation framework component of a reconfigurable scheduler. Smith and Becker [13] have created a fairly extensive ontology, and we aim to create a reconfigurable scheduler that can both represent and schedule as extensive a set of scheduling concepts as possible.

2.3 Evolutionary Scheduling: Previous Work

Genetic algorithms (and, more generally, evolutionary algorithms) have achieved success in scheduling applications [15]. There are several reasons why evolutionary algorithms are well suited for most scheduling problems, including many for which traditional mathematical programming techniques are inadequate. First, they are easy to apply to almost any optimization problem, including those with complex and/or discontinuous constraints/criteria that may derail other algorithms. Second, evolutionary algorithms are good at searching large and rugged search spaces to find nearly optimal solutions; furthermore, they can find good, though suboptimal, solutions very quickly. Third, genetic algorithms, with their population-based approach, allow for easy and effective large-scale parallelization [16], and this can provide a further performance boost.

The combination of complex constraints with the fact that orderings (rather than binary or numerical values) are often the primary outputs means that the standard chromosome representation (a binary string) is often not appropriate for scheduling problems. In fact, the first use of genetic algorithms for scheduling (by Davis [17]) also introduced one of the first non-standard chromosomes. There has been a large variety of representations and genetic operators used for evolutionary scheduling. Many of these are targeted to specific scheduling problems, such as the vehicle routing problem with time windows [18]. Such an approach is necessary in order to compete in terms of performance with other techniques tuned to specific problems, whether they be classic scheduling problems or real-world applications.

However, approaches targeted to specific scheduling problems are not useful for reconfigurable scheduling because of their lack of generality. An alternative is an order-based genetic algorithm combined with a greedy schedule builder [19, 20].

This is one of the earliest approaches to evolutionary scheduling, and its advantage is its universal applicability. The order-based genetic algorithm [21, 22] was developed based on the recognition that for problems like the traveling salesman problem, the goal is to find the best ordering of N objects. Its chromosome is a direct representation of a permutation of N objects, labeled 1 through N, and its operators are designed to manipulate chromosomes of this type. Whitley [20] and Syswerda [19] developed an approach whereby an order-based genetic algorithm can be applied to more complex types of scheduling problems by adding a greedy schedule builder. The order-based genetic algorithm generates orderings of the tasks to schedule, while the greedy schedule builder translates these orderings into schedules, handling the tasks in the order in which they are presented in the chromosome. This approach is universally applicable, since it is generally easy to create a greedy schedule builder for a scheduling problem.

3 Vishnu Overview

Vishnu is more complex than the standard scheduling application because in order to achieve its generality it needs to handle many different aspects of scheduling. In this section, we provide a general overview of the approach before exploring many of the details in the next section.

3.1 The Genetic Algorithm

In our approach, the genetic algorithm generates task orderings and relies on a schedule builder to translate these into actual schedules. The genetic representation uses an order-based chromosome. Each chromosome is some permutation of the integers 1 through N, where N is the number of tasks to schedule and each number corresponds to a task.

The only novelty of our genetic representation is that it incorporates *prerequisite constraints*. If task A has task B as a prerequisite, then task B must be handled earlier in the scheduling process than task A. (Note that this does not necessarily preclude task A from being scheduled at an earlier time than task B.) The genetic algorithm enforces prerequisite constraints by only generating chromosomes with orderings that obey all such constraints. A reordering operation is applied to every chromosome produced (either by mutation and crossover or during initialization of the population) to maintain these constraints. It works by finding any task that is earlier in the chromosome than any of its prerequisites and changing its location so that it is directly after the last of its prerequisites.

The crossover operator used by the genetic algorithm is position-based crossover [19], and its operation is illustrated in Figure 1. It works as follows. A set of positions is randomly selected (which in the example of Figure 1 are positions 4, 6 and 7). The elements at these selected positions in the first parent (which in the example are the integers 4, 6 and 7) are maintained at these positions in the child. The remaining elements (which in the example are the integers 1, 2, 3 and 5) are used to fill in the
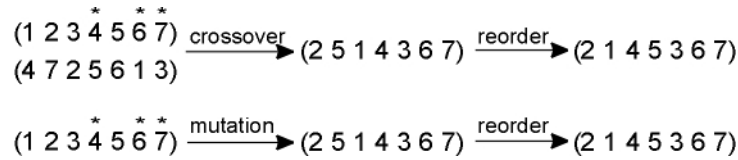
$$(1\ 2\ 3\ \overset{*}{4}\ 5\ \overset{*}{6}\ \overset{*}{7}) \xrightarrow{\text{crossover}} (2\ 5\ 1\ 4\ 3\ 6\ 7) \xrightarrow{\text{reorder}} (2\ 1\ 4\ 5\ 3\ 6\ 7)$$
$$(4\ 7\ 2\ 5\ 6\ 1\ 3)$$

$$(1\ 2\ 3\ \overset{*}{4}\ 5\ \overset{*}{6}\ \overset{*}{7}) \xrightarrow{\text{mutation}} (2\ 5\ 1\ 4\ 3\ 6\ 7) \xrightarrow{\text{reorder}} (2\ 1\ 4\ 5\ 3\ 6\ 7)$$

Fig. 1. The crossover and mutation operators. The *'s indicate the randomly selected positions. It is assumed that the prerequisites are such that task 4 precedes task 5 and task 2 precedes task 6.

remaining slots in the child, but will in general be at different positions in the child than in the first parent. The order of these elements in the child will be the same as their order in the second parent (which in the example means that 2 is placed in the first empty position, followed in order by 5, 1 and 3).

Also illustrated in Figure 1 is the mutation operator. It works the same as the crossover operator except without a second parent to provide the ordering for the subset of elements that are reordered in the child. Instead, the new order of the shuffled elements is randomly selected.

Each member of the initial population is generated by selecting a random ordering and then reordering the entries to obey the prerequisites constraints. The flow of operations of the genetic algorithm is shown in Figure 2.
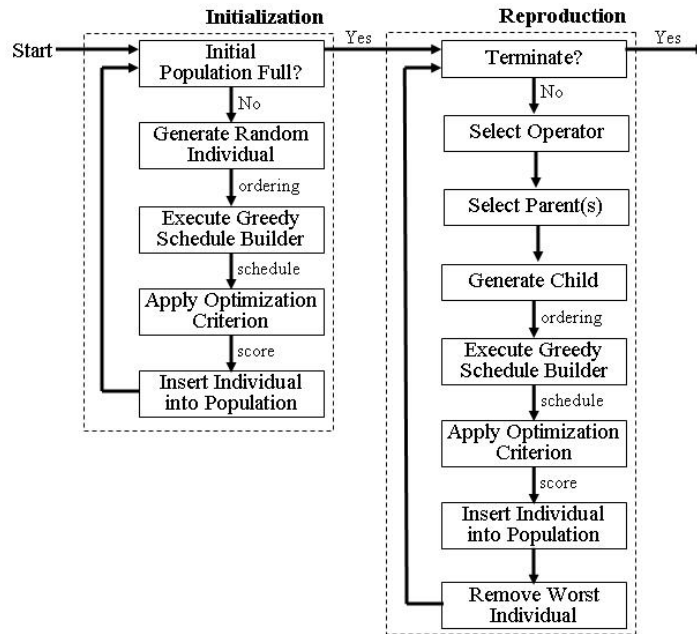

Fig. 2. The operation of the genetic algorithm.

The genetic algorithm is steady-state, which means that it generates and replaces one individual at a time rather than an entire population. The advantage of a steady-state replacement strategy is that the search generally proceeds faster, since the genetic algorithm can use good individuals as soon as they are created rather than waiting for generational boundaries. Since there are no generations, the amount of work done by the search algorithm is measured by the number of individuals evaluated. There is a uniqueness constraint to ensure that there are no two identical individuals in the population; duplicates generated by the genetic operators are discarded without being evaluated.

A fitness function is used to evaluate each individual, i.e. each task ordering. The evaluation starts by feeding the tasks to the greedy schedule builder in the specified order. The result is a schedule that obeys the required hard constraints. The optimization criterion, specified in a manner discussed in Section 4.3, then produces a numerical score representing the quality of the schedule, and hence the fitness of the individual.

There are two primary parameters for the user to specify: the population size and the number of evaluations. Increasing the population size and the number of evaluations increases the expected quality of the schedule at the expense of a longer search. As an extreme, if the user wants to execute just a greedy scheduler and bypass the genetic search, he can set the number of evaluations to be one.

Other parameters for the genetic algorithm are usually just set to their default values, as we do for all the experiments described in Section 5. The default probability of selecting mutation as the genetic operator is 0.5, with default probability also 0.5 for crossover. Parent selection is done using an exponential probability distribution, i.e. the individuals in the population are ranked and the ith best individual has selection probability that is some factor k as great as the $(i - 1)^{st}$ best. This factor k is set by default to be 1-(10/popSize). There are also parameters that can specify termination criteria that are alternatives to ending after a certain number of evaluations. These include: the maximum number of duplicates (which specifies to stop the search after a certain number of duplicate individuals has been discarded), the maximum run time (which specifies to terminate after a certain amount of wall clock time has elapsed), and the maximum age of the best individual (which specifies to terminate if no progress has been made during a certain number of consecutive evaluations). By default, these parameters are set so that the number of evaluations is guaranteed to be the termination criterion.

Note that it is possible to use other optimization techniques, such as simulated annealing or tabu search, to search the space of permutations (task orderings). Which technique is best depends on the characteristics of the problem, particularly the search space size and ruggedness. Genetic algorithms are a good match for Vishnu because they display good performance over a wide range of different search space characteristics, which is important because Vishnu should be able to handle a wide variety of different scheduling problems.

3.2 Greedy Schedule Builder Overview

Our greedy schedule builder needs to be more general than those designed for specific scheduling problems, such as the active schedule generation algorithm for job-shop scheduling [23]. While the details of the greedy schedule builder are complicated, the basic idea is simple. We provide a high-level description of the algorithm with italicized concepts corresponding to some (but not all) of the logic hooks where the user specifies the logic in a manner explained below.

The schedule builder assigns tasks one at a time in the specified order. For each task, it looks for the best resource(s) and time for that task, where the goodness of a resource and time is evaluated based on a specified *greedy criterion*. The schedule builder only considers resources *capable* of performing the task. For each capable resource, it checks the legality of scheduling the task at its specified *target start time*. (The target start time can be the beginning of time, indicating to schedule the task as early as possible.) If assigning at this time would not violate any hard constraints, such as *availability* or *capacity*, then it uses this time as the sole potential assignment time for this resource. If assigning at the target start time would violate a hard constraint, then the schedule builder finds the two times closest to the target time, one earlier and one later, at which it is legal to assign the task. Because one or both of these times may not exist, this search results in 0, 1 or 2 possible assignment times for this resource. From all the possible resources and corresponding times, the schedule builder selects the best one based on the greedy criterion and schedules the task there. If no legal resource and time exist (i.e., there is no way to schedule the task without violating a hard constraint), then this task is not scheduled.
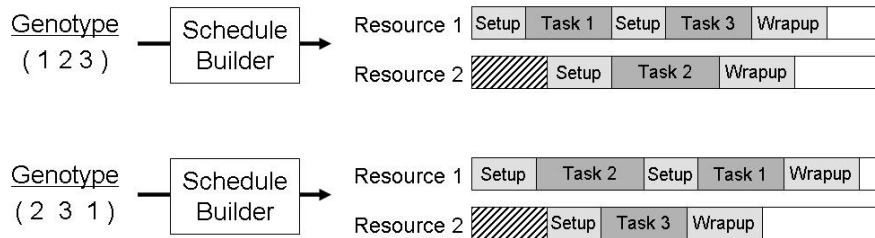

Fig. 3. The greedy schedule builder converts two different genotypes (task orderings) into two different phenotypes (schedules). The diagonally striped area indicates where the resource is unavailable.

Figure 3 illustrates the operation of the greedy schedule builder for a very simple scheduling problem. It shows how two different genotypes translate into two different phenotypes/schedules. The example makes certain assumptions about what the user has specified for the scheduling logic. These include the following default behaviors: (a) the target start time for all tasks is the beginning of scheduling window, (b) all resources are capable of performing all tasks, (c) tasks are always available, (d) there is no multitasking, i.e. a resource performs only one task at a time, and (e) each task requires only one resource. It also assumes that the greedy criterion

specifies that given a choice between multiple times at which to schedule a task, the earliest is the best. For the first genotype, the greedy scheduler starts with task 1. For resource 1, the task can be scheduled at its target time. For resource 2, the task cannot be scheduled at its target start time, so the schedule builder looks forward in time to find the first available slot, which is right after the initial block when resource 2 is unavailable. (Looking backward in time yields nothing.) Since the greedy criterion says that earlier is better, the schedule builder chooses resource 1. Next, it schedules task 2. The first available slot on resource 2 is earlier than that on resource 1, so the schedule builder assigns task 2 to resource 2. Finally, using the same logic, task 3 is assigned to resource 1 in the slot after task 1. (Note that each task has three stages: setup, execution and wrapup, although one or two of these may require zero time.) For the second genotype, the schedule builder uses the same procedure, but a different order in which the tasks are scheduled leads to a different schedule. If the optimization criterion is makespan (i.e. minimize the end time of the final task completed), then the top schedule is better than the bottom one.
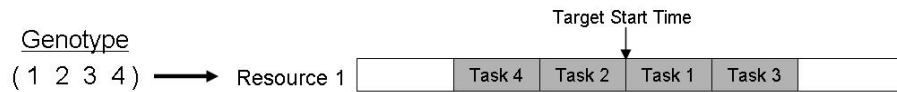

Fig. 4. Another greedy scheduler example, with different problem specifications.

Figure 4 shows what happens for a different set of problem specifications. The key differences from the prior one is: (a) the target start time is in the middle of the scheduling window rather than at the beginning, and (b) the greedy criterion is a penalty proportional to the deviation of the actual start time from the target time, with lateness penalized 1.5 times more heavily than earliness. The first task scheduled, task 1, is placed at the target time. The second task, task 2, cannot be assigned at the target time, so there are two options. Searching forward in time finds the first available slot is immediately after task 1, while searching backward finds the slot right before task 1. The greedy criterion selects the earlier slot, since the penalty is smaller. For task 3, the greedy criterion prefers the slot after task 1 rather than before task2.

3.3 Problem Representation Framework Overview

Like the greedy schedule builder, the problem representation framework operates on a simple basic concept, with the complexity in the details. We provide an overview here and fill in the details in the next section.

The genetic algorithm and greedy schedule builder both export logic hooks where the user can specify logic specific to a particular scheduling problem. Examples of such hooks include one that allows the user to specify the task execution duration for any task, one that allows the user to specify the prerequisites for any task, and one

that allows the user to specify the optimization criterion. The user defines a formula for each hook or accepts the default value for that hook.

The formula is then evaluated within context to provide the required information to the scheduling algorithm. Establishing the context primarily involves defining special variables and setting their values accordingly. The two most commonly used variables are *task* and *resource*, which refer to the particular task and/or resource about which to compute the information. All hooks provide the context variables *tasks* and *resources*, which are lists of all the tasks and resources, and *winStart*, which gives the earliest time in the scheduling window. The scheduling algorithm sets the context and evaluates the formula in that context, so all the user has to do is define the formula.

As an example, consider the Execution Duration hook, which specifies the amount of time that a resource spends executing a task and is discussed further in Section 4.3. We examine some different possibilities for formulas for this hook, starting with simple ones and then more complex ones.

- *Empty* - The default for this hook is 0, so if no formula is specified, then all tasks require 0 seconds to execute.
- **5** - This simple formula specifies that every task requires 5 seconds to execute.
- **task.duration** - This specifies that the time to execute a task is given by a field called *duration* in the task. (Note that the '.' character is the notation used to access a field inside a structure.)
- **task.distance / resource.speed** - The execution duration is the task's distance divided by the resource's speed.
- **entry (resource.durations, task.type)** - Each resource has a field *durations* that is a list of how long it takes that resource to execute different types of tasks, and each task has a numerical field that specifies its type. The task execution duration is the entry in the resource's list corresponding to the type of the task. (Note that *entry* is one of many predefined functions in Vishnu's formula language; it accesses the nth element of a list.)

4 Vishnu Details

In this section, we provide details about Vishnu. We start with a discussion of how to represent data, most notably data about the tasks and resources. We then describe the formula language and how formulas are evaluated. We next examine in-depth the different logic hooks and how the scheduling algorithm incorporates them. We conclude with examples of how to represent, and hence solve, three classic scheduling problems using Vishnu.

4.1 Scheduling Data

Vishnu provides a small number of atomic data types plus the ability to combine these atomic data types into composite data types. Some commonly used composite

data types are predefined, but each scheduling problem requires the definition of new problem-specific composite data types.

The atomic data types are *string*, *number*, *boolean*, and *time*, plus a special type, *list*, which is a variable-sized set of instances of a single specified data type. The predefined composite data types include *interval*, which contains the fields start (a *time*) and end (a *time*), and *matrix*, which contains fields numrows (a *number*), numcols (a *number*), and values (a *list* of *numbers*).

The problem-specific data types are built from the atomic and predefined types. The type for a field can itself be another problem-specific type, and hence it is possible to construct arbitrarily complex data types. For each scheduling problem, one data type must be declared to represent tasks and another type specified as representing resources. Each of these two types must have one field that serves as a key, providing unique identification of instances of this type.

All data for a scheduling problem must be instances of the composite data types (including the predefined types) defined for that problem. While the primary data (i.e. what the scheduler needs to schedule) are the tasks and resources, other types of data (e.g. business rules or distance matrices) may be required to define the scheduling logic.

Examples of representation of scheduling data are given in Section 4.4 and Section 5.

4.2 Formulas: Language and Evaluation

Formulas are built from the following types of components: constants, variables, accessors, operators, and functions. Accessors provide access to the fields of a data structure using the notation '.' followed by the field name. (For example, task.id gives the id field of the data structure referenced by the variable task.) There is a fixed set of standard arithmetic (+, -, *, /) and comparison operators (=, <>, <, <=, >, >=) written using infix notation. There is also an expandable library of functions, where the syntax for invoking a function is fcnName (*arg1*, ..., *argn*).

Note that *null* means no value, and all accessors, operators and functions must handle the case when one or more of their arguments are null, usually by just returning null. A null value can be introduced into a formula by context variables such as *next* or *previous* (which are null when the task of interest is respectively the last or first task for its resource) or by functions such as *resourceFor* and *taskStartTime* (which are null if the task is not assigned to a resource).

There are too many functions to list them all here, so we provide a representative sample:

- **if** (*a*, *b*, *c*) returns the evaluation of *b* if *a* evaluates to true and returns the evaluation of *c* if *a* is false, or returns null if the third argument, *c*, is omitted.
- **list** (...) combines all of its arguments into a list.
- **interval** (*startTime*, *endTime*) returns an interval object.
- **entry** (*list*, *index*) returns the element of *list* at *index*.
- **distance** (*location1*, *location2*) returns the distance between the two locations.

- **max** (*a*, *b*) returns the maximum of the two numbers.
- **and** (...) returns true if all of its boolean arguments are true.
- **hasValue** (*a*) returns false if *a* is null and true otherwise.
- **withVar** (*varName*, *varValue*, *a*) evalutes *a* with variable *varName* bound to *varValue*.
- **mapOver** (*list*, *varName*, *a*) binds the variable *varName* to each element of *list* in succession and evaluates *a*, returning the results as a new list.
- **sumOver** (*list*, *varName*, *a*) does the same as mapOver except returning the sum of the results.
- **taskStartTime** (*task*) returns the currently assigned start time of *task* (or null if *task* is not assigned). The functions **taskSetupTime**, **taskEndTime** and **taskWrapupTime** return the other three times associated with a task assignment. The functions **formerSetupTime** and **formerWrapupTime** provide the setup and wrapup times of an already scheduled task prior to the assignment of another task that may have caused these times to change.
- **resourceFor** (*task*) returns the currently assigned resource of *task*.
- **lastTask** (*resource*) returns the last task assigned to *resource*, and **complete** (*resource*) returns this task's end time.

Note that these functions can be classified into different types including mathematical functions (max, and, distance), control functions (if, withVar), data constructors (list, interval), list operations (mapOver, sumOver, entry), and schedule accessors (taskStartTime, taskEndTime, resourceFor).

Many examples of formulas are given in Sections 4.3, 4.4 and 5.

When the formulas are originally written, a compiler transforms them into parse trees, which are what the scheduler executes to evaluate formulas. As part of the compilation process, the compiler checks that each formula is correct with respect to data types, i.e. that the formula returns the data type expected by the hook and that each argument to each function is of the expected type. Before execution of a formula, the scheduler ensures that all context-dependent variables, such as *task* and *resource*, are set appropriately.

Since the formulas are evaluated many times in different contexts, the efficiency of formula evaluation is a major aspect of scheduler performance. One important way in which we have made formula evaluation more efficient is by caching (i.e. storing) the results of formulas and recalling, rather than recomputing, the results in the future. The trick is knowing when, both in terms of which formulas and which contexts, caching results and then recalling them is valid. The validity of cached results depends on the variables and functions used in the formula. If the formula contains any schedule-dependent function (i.e. a function whose returned value depends not just on its arguments but also the current schedule, such as taskStartTime or resourceFor), then the formula needs to be re-evaluated whenever the schedule or arguments change. If the formula contains no schedule-dependent functions but refers to both task and resource variables, it needs to be evaluated once for each task/resource pair. If the formula just references the task (or resource) variable, then the formula must be evaluated just once for each task (or resource).

Hook	Default	Description
Optimization Criterion	0	Measure of quality of current schedule
Greedy Criterion	0	Measure of quality of assignment of **task** to **resource**
Target Start Time	winStart	Optimal time for **task** to begin when assigned to **resource**
Prerequisites	empty list	Tasks that must be scheduled before scheduling **task**
Execution Duration	0	Seconds required for **resource** to perform **task**
Setup Duration	0	Seconds **resource** prepares for **task** after doing **previous**
Wrapup Duration	0	Seconds **resource** cleans up after **task** before doing **next**
Breakable	false	Can a task be executed in discontinuous time intervals?
Resource Unavailable	empty list	All intervals of time when **resource** is busy
Task Unavailable	empty list	All intervals when **task** cannot be scheduled on **resource**
Capability	true	Can **resource** perform **task**?
Capacity Thresholds	empty list	Maximum capacity of each type for **resource**
Capacity Contributions	empty list	How much **task** adds to each type of capacity of **resource**
Capacity Resets	empty list	Capacity restored to **resource** by performing **task**
Multitasking Type	none	How resources perform more than one task at a time
Groupable	false	Can **task1** and **task2** can be performed in the same group?
Multiresource Reqts	empty list	Set of requirements **task** needs satisfied by its resources
Satisfied Requirements	empty list	Contribution of **resource** to requirements of **task**
Auxilliary Tasks Before	empty list	Set of auxilliary tasks to schedule before scheduling **task**
Auxilliary Tasks After	empty list	Set of auxilliary tasks to schedule after scheduling **task**

Table 1. The scheduling logic hooks, with context variables bolded.

4.3 Hooks and Scheduling Logic

In this section, we discuss all of the hooks currently available in Vishnu. To define the logic for a scheduling problem, a user must associate a formula with each relevant hook or accept the hook's default. Table 1 provides a brief overview of the different hooks. Sections 4.3-4.3 discuss in greater detail the function of each hook and its role in the scheduling process. The hooks are divided into different classes to facilitate our explanation.

[Note that this set of hooks is only a current snapshot. Over time, we have gradually added new hooks, which explains why there is greater functionality available now than when we first introduced Vishnu in [3]. When we identify aspects of a scheduling problem that cannot be represented by the current hooks, we may specify a new hook to provide this functionality (along with updates to the schedule builder to utilize this hook). Careful consideration is given to ensure that a new hook is broadly applicable and not just a one-of-a-kind fix.]

Scheduler Directives

The first set of hooks we examine are those that instruct the scheduler how to execute rather than specifying problem constraints.

Optimization Criterion is the hook whose formula produces a numerical score for the current schedule. As discussed above, the genetic algorithm searches for the schedule with the smallest possible value for this score. All the different types of *soft constraints* of the problem must be aggregated into a single score, with the user (i.e. the person defining the problem) responsible for specifying how to combine them. The simplest and most common approach for combining penalties from different soft constraints is a weighted sum, but there are many other possibilities. Since this hook references the schedule as a whole rather than a particular task or resource, the only context variables are the standard ones: *tasks*, *resources* and *winStart*. A simple example is the formula that implements makespan, which scores a schedule based on the latest end time of any task:

$$\mathrm{maxOver}(tasks, "t", \mathrm{taskEndTime}(t)) - winStart$$

Note that subtracting winStart allows the formula to return a number, which is the expected data type, rather than a time. A more complicated sample formula combines a penalty for a task being late, a penalty for the time resources spend setting up, and a penalty for each task that was not scheduled:

$$\mathrm{sumOver}(tasks, "t", 10 * \mathrm{if}(\mathrm{taskStartTime}(t) > t.dueDate,$$
$$\mathrm{taskStartTime}(t) - t.dueDate, 0) + (\mathrm{taskStartTime}(t) - \mathrm{taskSetupTime}(t)) +$$
$$\mathrm{if}(\mathrm{hasValue}(\mathrm{resourceFor}(t)), 0, 1000000))$$

assuming that the task data type has a field *dueDate*. Other examples of schedule characteristics that might be penalized include resources working overtime or changes from a previous schedule. There should always be a formula specified for this hook except when the genetic algorithm is not used (with the user content to accept the first schedule produced by the greedy schedule builder).

Greedy Criterion is the hook whose formula produces a numerical score for any potential assignment of a task to a resource. As discussed above, it is used by the greedy scheduler to compare different possible task assignments to select the best one, and is essentially equivalent to a dispatch rule. It is often, but not always, an incremental version of the optimization criterion. Like the optimization criterion, it can combine multiple subcriteria into a single score. The context variables *task* and *resource* refer to the task and resource being assigned. A simple example for the case when earlier is better is the formula:

$$taskEndTime(task) - winStart$$

Another sample formula penalizes both any deviation from the task's target start time and an assignment to any resource other than the task's best resource:

$$\mathrm{abs}(\mathrm{taskStartTime}(task) - task.bestTime) +$$
$$\mathrm{if}(resource.name = task.bestResource, 0, 1000)$$

Other potential types of criteria include minimizing the travel (setup) time for a task or finding the least full resource.

Target Start Time tells the greedy schedule builder the optimal point on a resource's schedule to try to assign a task. As described above, for a given resource, the greedy scheduler searches forward and backward from the best time for the first legal assignment times and only considers assignments at these (at most) two times. This approach limits the number of assignment times to consider and hence the computation required. The context variables are *task* and *resource*. Often, the formula for this hook is not specified, instead accepting the default value *winStart*, which indicates to schedule the task as early as possible. An example of a case where earlier is not better is scheduling the delivery of food to a party. If the food arrives too early, it will not be fresh for the party, but it also should not arrive too late, hence making the best time n hours before the party. Note that a hard constraint on task availability (see below) may be used in conjunction to ensure that the food is never scheduled to be delivered after the party starts.

Prerequisites tells the genetic algorithm which other tasks must precede a given task in any generated task ordering, and hence will be handled earlier than this task in the greedy schedule building process. The context variable is *task*, and the formula must return a list of task names. The most common reason that task A would have task B as a prerequisite is that task A is constrained to start after task B finished, and hence needs task B scheduled to determine its own availability. However, there are other possibilities. For example, tasks A and B might need to be performed simultaneously, and task B is the more difficult of the two to schedule, so task B should be scheduled first and then task A assigned at the same time. Another example is backwards planning, where the last leg of a journey is scheduled first, with the scheduling process working backwards towards the earlier legs.

Task Durations

A second set of hooks defines the time required for a resource to perform a task. Recall that there are three stages for this process: setup, execution and wrapup. Each of these stages has a hook that tells the time in seconds required for this stage. Another hook tells whether or not task performance can be broken into multiple disconnected intervals.

Execution Duration determines the length of time spent in the execution stage when a resource performs a task. Since this potentially depends on the identity of both the task and resource, the context variables include *task* and *resource*. Some sample formulas for this hook were provided in Section 3.3. Note that during the execution stage, the task and resource both must be available, which limits the times at which the greedy scheduler can place the task.

Setup Duration computes the time a resource spends in the setup stage before performing a task. It generally depends not just on the task and resource but also on what the resource was previously doing, i.e. the previous task. Hence, the context variables include *task*, *resource* and *previous*, where *previous* references the previous task and is set to null if the task being performed is the earliest on the resource's schedule. For example, a painting machine might have a setup duration of 0 if the previous task has the same color as the current task and 2 minutes if the previous

(a) before adding Task 3

(b) after adding Task 3

Fig. 5. The setup and wrapup durations of already scheduled tasks can change when a new task is inserted.

task has a different color, as expressed by the formula:

$$\text{if}(previous.color = task.color, 0, 120)$$

Another example is where the setup time represents travel time between the previous and current tasks and is proportional to the distance between the geographic locations of these tasks, as given by the formula:

$$distance(task.location, previous.location)/resource.speed;$$

Because of the potential dependency on the previous task, the result of this hook may need to be recomputed as the greedy scheduler searches through the resource's schedule trying to find where the task fits. In this case, it also needs to recompute the setup duration of the following task when inserting in a spot other than the end of the schedule, as illustrated in Figure 5. Note that only the resource, and not the task, needs to be available during the setup stage.

Wrapup Duration computes the time a resource spends in the wrapup stage. It can depend on the task, the resource and what the resource is doing afterwards (i.e. the next task), so the context variables include *task*, *resource* and *next*. Consider the example where if a task is the final one on a resource's schedule, then the resource needs to spend five minutes cleaning up, but otherwise no time during wrapup. The formula to capture this is

$$\text{if}(\text{hasvalue}(next), 0, 300)$$

Like Setup Duration, the value for this hook is potentially recomputed not just for each potential position of the task on the resource's schedule but also for the preceding task already on the resource's schedule, as illustrated in Figure 5.

Breakable tells whether the task performance interval can be split into discontinuous sections. For example, if a coffee break for the resources is scheduled for 10:30-10:45, a breakable task that requires an hour can start at 10:00 and be completed in two intervals, 10:00-10:30 and 10:45-11:15. This hook is just a choice of two values, yes or no, with the default being no. (There could be further development in the future of the semantics for breakable tasks, e.g. allowing specification of the conditions under which tasks can be broken and into what size chunks the task can be broken.)

Availability

Another set of hooks specifies when tasks and resources are available to be scheduled.

Resource Unavailable specifies the times at which a resource is not available to be scheduled. This is independent of the tasks being scheduled, so the only context variable is *resource*. The formula returns a list of intervals. For example, if a person works from noon to 8PM on weekdays, then this resource is unavailable on weekends and between 8PM and noon. Other examples are fixed break times or maintenance downtimes.

Task Unavailable specifies the times at which a task is not available to be scheduled. Unavailable intervals can be independent of how other tasks are scheduled. For example, a service call can only be scheduled when a person is around to allow entrance, or a delivery cannot be scheduled after the time when the item is needed. However, unavailable intervals can also depend on other task's assignments. For example, it is very common that one task cannot be scheduled to start earlier than the end time of another task. Another example is that a task may only be allowed to be scheduled at the same time as another. A sample formula that specifies both that the task finishes before a due date and that it must wait for the end of another task is:

$$\text{list(interval}(task.dueDate, endTime), \text{interval}(winStart,$$
$$\text{if}(\text{hasValue}(task.followsTask),$$
$$\text{taskEndTime}(\text{taskNamed}(task.followsTask)), winStart)))$$

Capability specifies whether a particular resource is capable of performing a particular task. For example, an electrician can perform electrical wiring tasks but not plumbing tasks, and a painting robot can perform painting tasks but not welding tasks. The context variables are *task* and *resource*, and the hook returns a boolean indicating whether the resource is capable. An example formula that searches for a particular skill on a resource's list of skills possessed is

$$\text{contains}(resource.skills, task.skillRequired)$$

Capacities

Capacities are hard limits on what resources can do based on accumulation of quantities over multiple tasks. There are a few hooks for specifying the functionality of capacity constraints.

Capacity Thresholds is the hook that specifies for a resource the thresholds that cannot be exceeded for each type of capacity. In general, capacity has multiple dimensions. For example, there are limits on both total volume and total weight for the cargo transported by a vehicle. As another example, an employee may have limits on both the hours worked in a day and the hours worked in a week. Therefore, the hook expects a list of numbers that are the thresholds, as in the formula

$$\text{list}(resource.maxWeightCargo, resource.maxVolumeCargo)$$

The context variable is *resource*. If there are no capacity constraints, then the hook should use the default, which is the empty list.

(a) task ordering = (1 2 3 5 4)
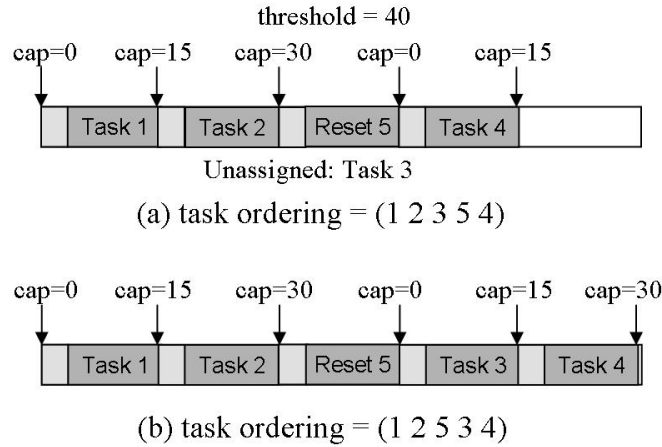

(b) task ordering = (1 2 5 3 4)

Fig. 6. This examples illustration how capacity can affect scheduling. The reset task, task 5, must be scheduled prior to other tasks to provide the additional capacity for these other tasks.

Capacity Contributions specifies the amount that a task contributes towards filling the capacity of a particular resource. A task cannot be assigned to a resource if it causes the threshold to be exceeded in any dimension. The context variables include *task* and *resource*, and formula should return a list of numbers of the same size returned by the Capacity Thresholds hook. The contributions of multiple tasks are accumulated by a simple vector sum. Note that when multitasking, i.e. a resource performing more than one task at a time, the capacities are summed and compared at a particular time, but otherwise they are accumulated over the duration of a resource's schedule. Continuing the weight and volume example, a sample formula is

$$\text{list}(task.weight, task.volume)$$

A more complex sample formula indicates that the amount of fuel required for a fuel truck to service a fuel request is the amount of fuel requested plus the amount of fuel for the truck to travel:

$$\text{list}(task.requestedFuel + resource.consumptionRate *$$
$$(\text{taskStartTime}(task) - \text{taskSetupTime}(task)))$$

Capacity Resets specifies when a task causes the accumulated capacity contributions of a resource to be reduced rather than increased. In many cases, the capacity contributions do not just keep on accumulating over time. At some point, an event happens that causes the accumulated counts to be reset, or partially reset. For example, with the capacity constraint that an employee can only work so many hours in a day, the total hours worked is reset to zero when a new day begins. Similarly, the total weight of a vehicle's cargo is reset to zero when all the cargo is dropped

off at a central depot. The formula on this hook returns a list of quantities by which to reduce the accumulated contributions, with the stipulation that they cannot be set to less than zero. The context variables are *task* and *resource*, and the default is an empty list, indicating no reset. An example formula that resets the contributions to zero if the task is to drop off the cargo is

$$\mathrm{if}(task.isDropoff, \mathrm{list}(1000000, 1000000), \mathrm{list}(0, 0))$$

Figure 6 shows the different aspects of capacity, including capacity resets, being used.

Multitasking

Multitasking is when a resource can perform multiple tasks simultaneously. There are a few hooks that determine this functionality.

Multitasking Type is a hook that has no formula but instead just a choice among three options. The first option is *none*, which means that the resource can only perform a single task at a time. This is the default and the most common option. With no multitasking, for the greedy scheduler to place a task on a resource's schedule, it needs to find a place where the time to perform the task, including the setup and wrapup, does not overlap the time to perform any other task already on the schedule. The second option is *grouped* (or batched) multitasking. In this case, a resource can perform multiple tasks simultaneously but only if all the tasks start at the same time and end at the same time. For example, consider a ship transporting cargo from one port to another. If each task is an item to transport, then the ship can perform more than one task at a time, but only if the items all depart from the same origin at the same time and travel to the same destination. The capacity constraints set the limit on how many simultaneous tasks a resource can handle. When assigning a task to a resource the greedy schedule builder can either add the task to an existing group or start a new group. The third option is *ungrouped* (or asynchronous) multitasking. In this case, the resource can perform multiple tasks simultaneously, and there is no need to synchronize the start and end times of the tasks. Hence, partial overlapping of the task performance periods is permitted as long as the capacity constraints are not violated at any time in the new task performance interval. An example of ungrouped multitasking is when a resource is actually a pool of homogeneous sub-resources, e.g. a set of N electricians or M cutting machines.

Groupable is a hook that only applies when using grouped multitasking. It specifies whether two tasks can be put in the same execution group, i.e. executed by the same resource at the same time. Context variables *task1* and *task2* provide references to the two tasks, and the formula returns a boolean. A sample formula indicates that two tasks are groupable if and only if they share the same origin and destination ports:

$$\mathrm{and}(task1.origin = task2.origin, task1.destination = task2.destination)$$

Multiresourcing

Multiresourcing is when tasks potentially require more than one resource, and this section describes the hooks that specify this functionality. Note that when a task uses more than one resource, the greedy scheduler currently assumes that all the resources are committed to the task for the entire duration of task execution. If the scheduling logic requires finer-grain control over the times at which the different resources are busy, then auxilliary tasks (see Section 4.3) should be used instead, with the single task split into multiple tasks that are tightly coupled.

Multiresource Requirements produces a list of numbers, similar to the capacity thresholds, that enumerate what requirements the resources in aggregate must satisfy. The context variable is *task*. As an example, if a class needs one classroom, one teacher, and a certain number of teaching assistants, the formula is

$$\text{list}(1, 1, task.assistantsRequired)$$

As another example, if a resupply task requires a certain number of gallons of fuels and a certain number of rounds of ammunition, the formula is

$$\text{list}(task.fuelRequired, task.ammoRequired)$$

The default is the empty list, and hence no multiresourcing.

Multiresource Contributions specifies a list of numbers, similar to the capacity contributions, that are the contributions that a resource makes towards satisfying a task's multiresourcing requirements. As resources are added to a task, the contributions accumulate until all requirements are fully satisfied or it is proven impossible to do so. The greedy scheduler adds a new resource to a set of resources potentially satisfying a task's requirements only if it provides a non-zero contribution to at least one of the requirements not yet satisfied. The context variables are *task* and *resource*. Continuing the class scheduling example, a sample formula is

$$\text{list}(\text{if}(resource.isClassroom, 1, 0), \text{if}(resource.isTeacher, 1, 0),$$
$$\text{if}(resource.isAssistant, 1, 0))$$

Auxilliary Tasks

Auxilliary tasks provide no direct benefit from being scheduled, and hence are not included in the task ordering created by the genetic algorithm. They are helper tasks that allow the primary tasks to complete. An auxilliary task that helps a particular primary task complete is scheduled at the same point in the scheduling process as the primary task. As an example, consider trying to schedule an air marshal to monitor a certain flight from Los Angeles to New York, but there is no air marshal scheduled to be in Los Angeles at that time. Scheduling a connecting flight as an auxilliary task can put a marshal in position to perform the primary task. As discussed above, auxilliary tasks are used for the situation when a task requires a variety of resources at different times. In this case, the task is divided into multiple tasks, one of which is

the primary task and the rest auxilliary. For example, a military air mission needs not just a plane and crew, but also maintenance crews and facilities, runways, airspace, etc. at different points in its execution.

Auxilliary Tasks Before provides a list of names of auxilliary tasks associated with a particular primary task that should be scheduled prior to scheduling the primary task. (Here, prior means earlier in the scheduling process, not earlier in the schedule.) The default is the empty list, and the context variable is *task*.

Auxilliary Tasks After is the same as Auxilliary Tasks Before but indicates the tasks scheduled subsequent to primary task rather than prior.

4.4 Examples - Classic Scheduling Problems

We now provide three examples of problem specifications using Vishnu. The problems are well-known and well-studied benchmarks from the operations research literature. (The OR-Library [24] is a good source of such classic problems and is available on the web at http://graph.ms.ic.ac.uk/info.html.) They are logically (though not computationally) simple, and therefore provide good examples with which to illustrate how to apply Vishnu before moving to more logically complex problems. While the Vishnu formulas can be hard to understand initially, it takes a relatively small amount of experience for a user to become acclimated to its method of problem representation.

Traveling Salesman Problem (TSP)

A salesman starts at a given city, travels to a set of other cities visiting each city once, and then returns to the starting city. The objective is to schedule the visits so as to minimize the total distance traveled.

Hook	Formula
Optimization Criterion	taskEndTime (taskNamed (”*City 1*”))
Prerequisites	if (*task.index* = 1, mapover (*tasks*, ”*t*”, if (*t.index* <> 1, *t.id*)))
Setup Duration	entry (*task.distances*, if (hasvalue (*previous*), *previous.index*, 1))

Table 2. Scheduling Logic for the Traveling Salesman Problem

The task data type, *city*, has fields id (a string), index (a number), and distances (a list of numbers). The resource data type, *salesman*, has the field id (a string). The data has one salesman with id = "Salesman". There are N cities. The ith city has index = i, id = "City i", and distances equal to a list containing the distance from each city to this one.

The hooks with associated formulas are shown in Table 2. The optimization criterion is the completion time, where this time is equal to the distance traveled. The prerequisities constraint indicates that the salesman cannot return to the city of origin, city 1, until he has visited every other city. The setup duration is obtained by looking up the distance from the previous city in the current city's list of distances.

Job-Shop Scheduling Problem (JSSP)

This problem was originally proposed by [25]. There are M machines and N manufacturing jobs to be completed. Each job has M steps, with each step corresponding to a different specified machine. There is a specified order in which the steps for a certain job must be performed, with each step not able to start until the previous step has ended. The objective is to minimize the end time of the last step completed.

Hook	Formula
Optimization Criterion	maxover (*resources*, "*r*", complete (*r*)) - *winStart*
Prerequisites	if (*task.preceedingStep* <> "", list (*task.preceedingStep*))
Execution Duration	*task.duration*
Capability	*task.machine = resource.id*
Task Unavailable	if (*task.preceedingStep* <> "", list (interval (*winStart*, taskEndTime (taskNamed (*task.preceedingStep*)))))

Table 3. Scheduling Logic for the Job-shop Scheduling Problem

The task data type, *step*, has fields id (string), duration (number), machine (string), and preceedingStep (string). The resource data type, *machine*, has field id (string).

The hooks with associated formulas are shown in Table 3. The optimization criterion is the makespan. If a task is not the first step in a job, its prerequisite is the step that precedes it, and it is unavailable to be scheduled earlier than the end time of this preceding task. Only the designated machine is capable of performing a task. Since there is only one choice of resource, there is no need for a greedy criterion.

Hook	Formula
Optimization Criterion	sumOver (*tasks*, "*t*", (taskStartTime (*t*) - taskSetupTime (*t*)) + if (hasvalue (resourceFor (*t*)), 0, 1E7))
Greedy Criterion	if (hasvalue (*next*), 0, if (hasvalue (*previous*), taskWrapupTime (*task*) - formerWrapupTime (*previous*), taskWrapupTime (*task*) - *winStart*))
Prerequisites	mapover (*tasks*, "*t*", if (*t.latest* < *task.earliest*, *t.id*))
Execution Duration	extra.serviceTime
Setup Duration	distance (*task.location*, if (hasvalue (*previous*), *previous.location*, *extra.depotLocation*))
Wrapup Duration	if (hasvalue (*next*), 0, distance (*task.location*, *extra.depotLocation*))
Task Unavailable	list (interval (*winStart*, *task.readyTime*), interval (*task.latest + extra.serviceTime*), *endTime*))
Capacity Contributions	*task.load*
Capacity Thresholds	*extra.capacity*

Table 4. Scheduling Logic for the Vehicle Routing Problem with Time Windows

Vehicle Routing Problem with Time Windows (VRPTW)

This is a more logically complex problem than the previous two. It is described in [26]. There are M vehicles and N customers from whom to pick up cargo. Each vehicle has a limited capacity for cargo, and each piece of cargo contributes a different amount towards this capacity. There is a certain window of time in which each pickup must be initiated, and the pickups require a certain non-zero time. Each vehicle that is utilized starts at a central depot, makes a circuit of all its customers, and then returns to the depot. The objective is to minimize the total distance traveled by the vehicles.

The task data type has fields id (string), load (number), earliest (number), latest (number), and location (xycoor). The resource data type has fields id (string) and capacity (number). A third data type has fields serviceTime (number) and depotLocation (xycoor) and has a single instance named *extra*.

The hooks with associated formulas are shown in Table 4. The optimization criterion is the sum of the distances traveled by the vehicles plus a large penalty that is proportional to the number of unassigned tasks. The execution duration is just the constant defined in *extra*. The setup duration is the distance from the location of the previous pickup, or if this is the first pickup, the distance from the depot. The wrapup duration is nonzero only if this is the last pickup, in which case it is the distance to return to the depot. Each customer is unavailable before the start of its pickup window and after the end of this window allowing the time for pickup. The capacity formulas indicate that the sum of the loads contributed by the different customers cannot exceed a vehicle's capacity.

The greedy criterion and prerequisites formulas are actually not part of the problem specification but directives that help the scheduler find a solution faster. Task B is defined to be a prerequisite for task A if the end of B's pickup window is earlier than the start of A's window. This allows the greedy scheduler to predominantly build the schedule forward in time. The greedy criterion states that if there is a way to fit a new task to schedule earlier than the last task of a resource, that is the preferred assignment. Otherwise, the preference is to find the assignment that packs the schedule as compactly as possible.

4.5 Additional Scheduling Capabilities

While problem specification and automated schedule creation are the core capabilities required by a reconfigurable scheduler, there are additional capabilities that make it more practical and generally applicable. We now provide a brief discussion of some of these additional capabilities that Vishnu possesses.

Dynamic rescheduling is the process of creating a modified schedule from an existing schedule in response to updates to the data. Many real-world scheduling problems require the ability to change the schedule "on the fly", i.e. during the process of executing the schedule. Vishnu provides a few mechanisms to support dynamic rescheduling, including sticky assignments and frozen assignments [27].

In many cases, when doing dynamic rescheduling, the scheduler should attempt to minimize the perturbation from the prior schedule, i.e. maximize *schedule stability*. A *sticky* assignment is a type of soft constraint that penalizes a new assignment for a task for differing from the previous assignment for the task. To implement sticky assignments, the scheduler provides functions priorResource, priorStartTime, etc. that provide information about the prior assignment. This allows the user to explicitly include penalties for maintaining schedule stability on the Optimization Criterion, Greedy Criterion, and Target Start Time hooks. Furthermore, tasks provide data fields that indicate the level of commitment to the current assignment, providing a priority for maintaining this assignment.

The scheduler also allows an assignment to be *frozen*, i.e. creating a hard constraint that the assignment stay the same upon rescheduling. There are two reasons why a user might want to make an assignment frozen, which is a hard constraint, rather than sticky, which is a soft constraint. First, the human scheduler may want to force a particular assignment without giving an override option to the computer. Second, it is much more efficient computationally, since tasks with frozen assignments fall automatically into place without the need to search for the best position. In contrast, tasks with sticky assignments require a scheduling decision to be made, and in general place the same computational burden on the scheduler as a new and previously unscheduled task.

Schedule display and interactive scheduling are other features that enhances Vishnu's usefulness. Vishnu automatically generates color-coded Gantt charts to display a schedule. The colors used for the different assignments and the text to display are specified by the user employing formulas of the same type used to specify the scheduling logic. The display also includes the capability to generate user-defined spreadsheet-like data tables, with the data to display also based on formulas.

Vishnu not only displays the schedule for the user but also allows the user to modify the schedule. It provides various ways for users to make their own assignments, or undo existing assignments, including drag-and-drop. After a user has made his own assignment of a task to a resource, he can make this assignment sticky or frozen so that the scheduler cannot just discard it during the next round of scheduling. In this way, the user and automated scheduler can work together to produce a final schedule.

A **composable software architecture** makes a reconfigurable scheduler, such as Vishnu, applicable in more situations. For certain applications, a single-user standalone scheduler is sufficient, with any sharing of schedule data done via files. However, other applications require that the scheduler be integrated as part of a larger software system. Although a discussion of the details of the software architecture of Vishnu is beyond the scope of this paper, we do note that Vishnu is composable in a variety of ways. First, it has a web-based deployment mode, in which schedules are stored in a database and are accessible via a web server. This allows shared viewing and editing of schedules across multiple locations. Second, Vishnu has been integrated as the basis for scheduling agents in a multiagent scheduling architecture [28]. This allows multiple schedulers to cooperate on producing schedules, which

is important when a scheduling problem is too big and/or too heterogeneous for a single scheduler. Third, Vishnu provides standard interfaces for communication with non-scheduling software, including formats for passing data back and forth and a well-defined interface. This allows easy integration into larger software systems.

5 Evaluation

The traditional metrics for evaluating scheduling algorithms do not measure what Vishnu is trying to accomplish. Traditionally, evaluation involves selecting a set of benchmarks that are instances of the particular problem for which the scheduler was designed. The performance of the scheduling algorithm can be compared to what other algorithms achieve on the same benchmarks, both in terms of quality of solution and the time to reach the solution. This makes sense as a way to compare schedulers targeted to a specific problem.

However, this is not the right way to evaluate Vishnu. The goal of Vishnu is to make it quick and easy to develop optimized scheduling solutions to new scheduling problems. Therefore, the primary metrics should be (a) the ability to solve a wide range of problems with just reconfiguration and (b) the ease of solution development, i.e. the time and effort it takes a user to configure for a particular problem. For certain problems, particularly well-studied benchmarks, Vishnu's generality will mean that its performance based on traditional metrics will be inferior to that of problem-specific schedulers, sacrificing raw speed for flexibility and ease of solution development. For those readers familiar with software development, this tradeoff is analogous to that between high-level programming languages, such as C++ or Java, and low-level languages, such as assembly or microcode.

While there is no obvious way to measure the ease of development or range of problems solved, we can provide sample problems, anecdotes, and an analysis of the capabilities of Vishnu. Of course, traditional measures of scheduling performance still matter for Vishnu (just as for programming languages), so we additionally provide some performance numbers on a few benchmark problems to show that Vishnu passes the threshold for acceptability. Section 5.1 discusses two sample problems logically more complex than the benchmark problems as a way to show how easy it is to develop scheduler for new problems, even if they are complex. Section 5.2 provides speed and optimality performance numbers on some common benchmark problems. Section 5.3 provides a brief analysis of the capabilities of Vishnu not possessed by other reconfigurable schedulers, which allow it to solve a wider range of problems.

5.1 Sample Problems

The following two sample problems demonstrate how a Vishnu scheduler can be easily specified for problems with greater logical complexity than traditional benchmarks. These are just a representative set, and a variety of additional problems are available in the demonstration at the Vishnu web site [29].

Air Marshal Scheduling

Air marshals are people who fly on commercial airline flights and monitor them for terrorist, or other illegal, activities. There are not enough marshals to monitor all flights. Therefore, good scheduling is required to put marshals on as many flights, particularly those designated as high priority, as possible while not putting undue burdens on the marshals.

Each marshal has an airport designated as his home base. He should take a set of flights that form a circuit that eventually brings him back home. The **hard constraints** of the problem are

(a) All flights leave and arrive at their scheduled times.
(b) The marshal must be located at an airport to take a flight that flies from it.
(c) A marshal must arrive at least an hour early for a flight.
(d) A marshal can work at most 14 hours in a day, with a maximum of 8 hours spent on flights.
(e) A marshal cannot work during his time off.

The **soft constraints** are

(a) As many different flights as possible, particularly the high priority ones, should be covered.
(b) Marshals should return home at the end of the current scheduling window.

Air marshal scheduling is a variation of the air crew scheduling problem. Over time, a standard approach for air crew scheduling has emerged where the problem is split into two subproblems, crew pairing and crew assignment (or rostering) [30]. The former creates a set of circuits of flights, each of which ends at the same airport as it begins. The latter determines which crew members to assign to each route/circuit. This approach works well for large-scale problems when there are many flights and many potential crew members. However, it does not scale down well to small numbers of crew members (marshals), in which case building the routes cannot be done independent of knowledge of the crew members' work schedule.

We were able to create an optimizing solution to the air marshal scheduling problem within a day using Vishnu. This is very rapid turnaround time given the complexity of the problem. A version is shown in Table 5. A big benefit of Vishnu is the compactness of the problem representation, requiring only a small number of lines of formulas.

We now describe how the specifications in Table 5 meet each of the constraints for the problem given above, starting with the **hard constraints**.

(a) To ensure that flights only depart and arrive as scheduled requires a combination of formulas on two hooks. Execution Duration constrains each flight to extend for its scheduled length of time, while Task Unavailable constraints a flight to not start before its schedule departure or end after its scheduled arrival.
(b) To ensure that a marshal must be at an airport in order to fly from it involves two hooks. The Setup Duration hook determines at which airport the marshal is located by finding the arrival airport of the marshal's previous flight, or the marshal's home airport if the current flight is his first. If this airport is not the departure airport of the current flight, the formula on the hook evaluates to a

Hook	Formula
Optimization Criterion	sumOver (*tasks*, "*t*", if (hasValue (resourceFor (*t*)), 0, if (*t.priority* = 1, 1, if (*t.priority* = 2, 0.2, 0.04)))) + sumOver (*resources*, "*r*", if (lastTask (*r*).*arrivesWhere* = *resource.home*, 0, 0.5))
Greedy Criterion	*task.departsWhen* - complete (*resource*)
Prerequisites	mapover (*tasks*, "*t2*", if (*task.departsWhen* >= *t2.arrivesWhen* + entry (*t2.arrivesWhere.minConnectTime*, *task.departsWhere.index*), *t2.name*))
Execution Duration	*task.arrivesWhen* - *task.departsWhen*
Setup Duration	if (if (hasvalue (*previous*), *previous.arrivesWhere* = *task.departsWhere*, *resource.home* = *task.departsWhere*), 3600, 999999)
Resource Unavailable	resource.unavailable
Task Unavailable	list (interval (*winStart*, *task.departsWhen*), interval (*task.arrivesWhen*, *endTime*))
Capability	complete (*resource*) + entry (lastTask (*resource*).*arrivesWhere.minConnectTime*, *task.departsWhere.index*) <= *task.departsWhen*
Capacity Thresholds	list (28800, 50400)
Capacity Contributions	list (*task.arrivesWhen* - *task.departsWhen*, if (and (hasvalue (*previous*), not (*isReset*)), *task.arrivesWhen* - *previous.arrivesWhen*, *task.arrivesWhen* - *task.departsWhen*) + if (and (not (hasvalue (*next*)), *endTime* - *task.arrivesWhen* < 68400.0), entry (*task.arrivesWhere.travelTimes*, *resource.home.index*), 0))
Capacity Resets	if (and (hasvalue (*previous*), *task.departsWhen* - *previous.arrivesWhen* > 36000, andOver (tasksFor (*resource*), "*t*", or (capacityReset (*t*, 1.0) = 0.0, taskStartTime (*t*) >= taskStartTime (*task*)))), list (1E8, 1E8), list (0, 0))
Auxilliary Tasks Before	list (entry (entry (if (hasValue (lastTask (*resource*)), lastTask (*resource*).*arrivesWhere*, *resource.home*).*connectSchedules*, *task.departsWhere.index*).*latestConnect*, (*task.departsWhen* - *winStart*) / 1800 + 1))

Table 5. Scheduling Logic for the Air Marshal Scheduling problem

very large number, making it impossible to assign the flight to the marshal. The Auxilliary Tasks Before hook checks to see if there is a connecting flight in the case that the two airports are different. If so, it specifies to assign this flight as a way to position the marshal for the flight of interest.

(c) To ensure that marshals are at least an hour early for their flight, the Setup Duration evaluates to 3600 seconds when the marshal is at the right airport.

(d) Enforcement of the flight-time and working-hours limitations relies on the three capacity-related hooks. The limits are set to 8 hours and 14 hours respectively by the Capacity Thresholds hook. The Capacity Contributions hook specifies

that the additional flight time is the length of the flight, while the additional time worked is the length of the flight if this is the first flight since a reset or, otherwise, the length of the flight plus the time spent between flights.

(e) The marshals' time off from work is protected from assignments by the Resource Unavailable hook.

The Optimization Criterion formula embodies the **soft constraints**, penalizing 1.0, 0.4 and 0.02 for each flight not covered with priority 1, 2 and 3 respectively, plus 0.5 for each air marshal not home at the end of the scheduling window. Like in the vehicle routing problem discussed above, other hooks assist in achieving the goals of the Optimization Criterion. The Greedy Criterion hook specifies that the greedy scheduler should pick the marshal for a flight that would have the shortest waiting from his last flight to this flight. The Prerequisites hook says that a flight cannot be scheduled until all flights that could possibly feed it by bringing a marshal to its originating airport are all scheduled. At that point, the greedy scheduler knows which marshals will be at the airport and therefore can make an informed decision. The Capability and Capacity Contributions hooks both help get marshals home by prohibiting them from traveling too far away without enough time remaining to get home. Specifying this as a hard constraint rather than just a greedy preference makes it much less likely that a marshal will not make it home.

Battlefield Supply Scheduling

An agile military requires that its combat units be able to fight for long periods of time without running out of supplies. To accomplish this, supply trucks drive around the battlefield, although preferably not the area of actual fighting, delivering their cargo to the vehicles of the combat units. Each unit has multiple vehicles all geographically clustered, so it is efficient for one supply truck (or a small number of supply trucks) to deliver all the supplies of a particular unit. The problem is how to schedule the deliveries of multiple supply trucks to the different combat units.

There are actually different battlefield supply scheduling problems with different constraints for each broad class of supplies. Here, we consider two different classes of supplies: fuel and ammunition (ammo). The problem specifications shown in Table 6 show the formulas separately for food and ammunition for those hooks where the formulas differ. One of the big benefits of Vishnu is that it allows easy adjustment for the idiosyncracies of variations on a single problem, as illustrated by the ease with which we can specify different problem specification, and hence schedulers, for the two types of supplies.

The **hard constraints** of the problem are

(a) Each delivery requires five minutes to execute.

(b) The supply trucks travel a time equal to the distance between the points divided by the truck's average speed.

(c) There is a fixed window of time for each delivery to occur.

(d) (ammo) There are different types of ammunition, so each request needs to be matched with the types and quantities available on a supply truck.

Hook	Formula
Greedy Criterion	*omitted for brevity*
Target Start Time	*task.desiredTime* - 300
Prerequisites	mapover (*tasks*, "*t*", if (*t.DesiredTime ¡ task.DesiredTime, t.id*))
Execution Duration	300
Setup Duration	distance (if (hasValue (*previous*), *previous.location*, *resource.initialLocation*), *task.location*) / *resource.speed* * 3600
Capability (Fuel)	or (*task.refillAmount* = 0, *task.recipient* = *resource.id*)
Capability (Ammo)	and (or (*task.refillAmount* = 0, *task.recipient* = *resource.id*), hasValue (find (*resource.loadedAmmo*, "*a*", *a.type* = *task.desiredType*)))
Task Unavailable	list (interval (*winStart, task.earliest*), interval (*task.latest, endTime*))
Cap Thresholds (Fuel)	*resource.initialGallons*
Cap Thresh (Ammo)	mapOver (*resource.loadedAmmo*, "*a*", *a.rounds*)
Capacity Contributions (Fuel)	if (*task.refillAmount* > 0, 0, withVar ("f", find (*task.fuelProfile*, "f2", and (taskStartTime (*task*) $>=$ *f2.startTime*, taskStartTime (*task*) $<=$ *f2.endTime*)), (*f.endValue* - *f.startValue*) * (taskStartTime (*task*) - *f.startTime*) / (*f.endTime* - *f.startTime*) + *f.startValue*))
Capacity Contributions (Ammo)	mapOver (*resource.loadedAmmo*, "*a*", if (and (*task.refillAmount* = 0, *a.type* = *task.desiredType*), max (1, min (*task.desiredQuantity*, capacityRemaining (*resource, task.DesiredTime*, indexOf (resourceFor (*task*)*.loadedAmmo*, "*a*", *a.Type* = *task.desiredType*)))), 0))
Capacity Resets (Fuel)	*task.refillAmount*
Cap Resets (Ammo)	mapOver (*resource.loadedAmmo*, "*a*", if (*a.type* = *task.desiredType, task.refillAmount*, 0)))

Table 6. Scheduling Logic for the Battlefield Supply Scheduling problem

(e) (ammo) The requested amount is a maximum, and less can be delivered if the full amount is not available.

(f) (fuel) A different amount of fuel is required depending on when the fuel is delivered. The recipient continually consumes fuel while awaiting the delivery, and hence needs more fuel the later the delivery. A piecewise linear fuel profile specifies how much is required.

(g) The supply trucks themselves are restocked at fixed times and locations by fixed amounts.

The **soft constraints** are

(a) The schedule should fill as many requests as possible.

(b) For each delivery, the time should be as close as possible to the desired time with preference to earlier than later.

(c) The schedule should minimize travel time/distance of the supply trucks.

The problem specification in Table 6 satisfies the **hard constraints** as follows.

(a) The time for a delivery is specified by the Execution Duration hook.

(b) The travel time is specified by Setup Duration, with the prior location being either the location of the last delivery or, if this is the first delivery, then the truck's initial location.

(c) The delivery window is specified by Task Unavailable.

(d) (ammo) To ensure that the truck carries the right type of ammunition, the Capability formula checks that the desired type is in the truck's list of initial supplies. The quantities of the different types of ammunition are tracked using the capacities, with each dimension of the capacity corresponding to a particular type of ammunition. Capacity Thresholds indicates that the maximum of each type of supply is as given at initialization. Capacity Contributions removes the amount delivered from the truck's stock.

(e) (ammo) Capacity Contributions sets the amount delivered to be the minimum of the amount requested and the amount left on the truck but never less than one.

(f) (fuel) Each task has an associated fuel profile that is a list with each element corresponding to one piece of the piecewise linear function. Capacity Contributions finds the right piece and interpolates between the endpoints.

(g) Capacity Resets restores the inventory on the supply truck.

With respect to the **soft constraints**, for the particular application it was more important to have fast turnaround than fully optimized schedules. So, we used just a single greedy schedule generated from a random task ordering. This meant that we had no need for an optimization criterion. The Greedy Criterion is not shown in Table 6 because it is a bit too long and is not particularly instructive. However, it is simple in concept. There are four penalty terms. One penalizes additional travel time. A second term penalizes the deviation of the actual delivery time from the desired time, with a much heavier weight for being late. The third rewards putting two supply tasks from the same unit consecutively on a resource. The fourth penalizes putting two supply tasks from different units contiguously on a resource. The Prerequisites formula specifies to perform the scheduling of tasks in the order of their desired delivery times, although there is much room for randomness in the ordering with many tasks having the same desired time.

This was a successful application of Vishnu [31]. We were able to quickly build different types of supply schedulers and easily adjust them to changing specifications of the problem. We were also able to easily integrate the schedulers into a larger multiagent system for managing supplies.

5.2 Performance Results

We now provide performance numbers on some benchmark classic scheduling problems. As discussed above, Vishnu cannot always compete in terms of speed and optimality with problem-specific algorithms. However, the results on these benchmarks still provide some idea of how Vishnu will perform on other problems similar in scale that do not have existing solutions.

Problem Name	Population Size	Evaluations	Optimal Score	Median Score	Average Score	Avg Time (M:S)
JSSP-mt06	1000	5000	55	55	55	0:01
JSSP-mt10	500	10000	930	1012	1010	0:06
JSSP-mt10	5000	100000	930	982	982	1:04
JSSP-mt10	50000	1000000	930	961	962	10:21
TSP-bays29	5000	140000	2020	2028	2042	0:11
VRPTW-c101	100	200	827.3	828.9	828.9	< 0:01

Table 7. Summary of experimental results

Table 7 summarizes the results. All the experiments involved ten runs of Vishnu on the problem, reporting the mean and median scores of the resulting schedule produced and the mean time for the run to complete. In addition, the table shows the two parameters that need to be selected for the genetic algorithm, the population size and the number of evaluations performed, as well as the score for the known optimal solution to the problem. All the runs were made on a 2.8GHz Pentium 4 processor.

The data sets we used were

- Muth-Thompson 6x6 (mt06) and Muth-Thompson 10x10 (mt10) are two standard benchmarks for job-shop scheduling [25]. The former has 6 machines/resources and 6 jobs, and hence 36 tasks. The latter has 10 machines and 10 jobs, and hence 100 tasks. They are available from OR-Library (under the names ft06 and ft10).
- The bays29 traveling salesman problem is a 29-city symmetric problem available at the TSPLIB web site [32].
- The c101 vehicle routing problem with time windows is one of the Solomon benchmarks [26]. There are 100 pickups/tasks and 25 vehicles/resources. This is one of the instances where the time windows are very tight.

Job-Shop Scheduling Problem - The Muth-Thompson 6x6 problem is not a difficult problem, but it is also not trivial given the 36 tasks to schedule. Therefore, the ability of the automated scheduler to consistently find an optimal solution in under 5000 evaluations and 1 second reflects well on both the effectiveness of the genetic search algorithm (which needs to explore only a very small fraction of the search space of task orderings) and the efficiency of the greedy schedule builder (which despite its generality can still build each schedule in approximately 0.2 msec).

The Muth-Thompson 10x10 problem is much more difficult. In fact, despite much attention, there was no solution proven to be optimal until relatively recently [33]. The results show three sets of experiments with three different genetic algorithm parameters. The first set of parameters has a small population size, leading to fast convergence. The second set of parameters is a factor of ten longer before convergence than the first, and the third a factor of ten longer than the second. This illustrates two properties of our reconfigurable scheduler (and of many pure genetic algorithms). First, on difficult problems it can quickly find a reasonable solution (within

8.6% of the optimum in 6 seconds and within 5.6% of the optimum in a minute) but, even with much longer runs, may not find the optimum solution. Second, there is a tradeoff between the search time and the expected quality of the solution, which can be selected explicitly via choice of the parameters.

Traveling Salesman Problem - The 29-city problem used is a small one, but it is sufficient to show that our algorithm clearly cannot compete with custom designed algorithms. For example, the Concorde algorithm [34] completes the bays29 problem in 0.13 seconds, as compared to our algorithm taking over 11 seconds on a much faster machine. The fact that our algorithm is a genetic algorithm is not the primary issue, as a variety of competetive genetic algorithms for the traveling salesman problem attest [35]. There are two main reasons for our algorithm's shortcomings, both of which are related to the fact that the traveling salesman problem is purely a routing problem rather than a true scheduling problem (insofar as time-based constraints are not involved). First, there is a lot known about the structure of the search space (particularly when the distances are symmetric), and large performance gains can be achieved by designing an algorithm that exploits this structure. Second, there is a large software overhead, since Vishnu builds a full schedule as part of the evaluation process, while a particular route can be evaluated just by summing the distances.

Vehicle Routing Problem with Time Windows (VRPTW) - The performance of our algorithm on this problem is respectable, scheduling all 100 tasks in a nearly optimal fashion in less than a second. Note that the genetic search required little work, one pseudo-generation, beyond the initial population. The use of a steady-state genetic algorithm helps the search proceed this quickly, since the best children can immediately produce their own offspring. Even more important to the performance is the fact that the Prerequisites and Greedy Criterion formulas were specified so as to tightly pack the schedule.

Overall Conclusions - The experimental results do support the premise that our reconfigurable scheduler can provide reasonable performance on a range of problems. The poor performance on the traveling salesman problem is, in some sense, the exception that proves the rule. Not only did it require many researcher-years to discover the much better solutions (which is a magnitude of effort that cannot be devoted to every scheduling problem), but more importantly it is also a highly "atypical" scheduling problem because of its very simple constraints, the lack of any concept of time, and the existence of exploitable structure in its search space.

5.3 Analysis of Capabilities

Comparing different reconfigurable schedulers with regards to their generality and flexibility to solve a large variety of problems is an inexact and difficult task. There are no sets of benchmark problems where we can say that if a reconfigurable scheduler can solve all problems in set X then it achieves level Y of reconfigurability. There is a wide range of different types of scheduling problem features that a reconfigurable problem should handle, many of which were discussed in Section 4.3 (multitasking, multiresourcing, capacities, etc.). At this point, the best that we can do

is list some of the capabilities that help distinguish Vishnu from other reconfigurable schedulers.

These features include

- Auxilliary Tasks - While their only use in the example problems was to represent connecting flights, their most powerful use is allowing the decomposition of a complex task into multiple subtasks with different requirements, yet handling them as a single entity in the scheduling process.
- Capacity Resets - Capacities are a more fluid concept than just adding up the individual contributions and comparing to a threshold, and resets are an important part of this extra complexity.
- Dynamically Computed Constraints - Quantities such as task setup and execution times, capabilities, task availabilities, and capacity contributions all can depend on the schedule and can change as the schedule gets built.
- Algorithmically Specified Constraints - Constraints specified using a real, though simple, programming language can express a wider range of possibilities than the mathematically specified constraints of mathematical programming or just picking among some prespecified types of constraints.
- Scheduler Directives - These allow the user to guide the scheduler to produce better solutions faster, and do so in an easily understandable way and within paradigm of the problem specification framework. The only scheduler controls that are not part of this framework are the the population size and number of evaluations, which the user sets by selecting values rather than formulas.
- Additional Capabilities - Dynamic rescheduling, interactive scheduling, and software composability are all important for building real-world scheduling systems.

Even more important than the particular features already in Vishnu is that the infrastructure allows easy addition of new features and capabilities as required. If we encounter a scheduling problem that requires a capability not currently contained in Vishnu, we can add a new hook and modify the greedy scheduler to handle this new constraint. Since the greedy scheduler does not rely on any idiosyncratic search algorithm, it is generally the case that it can be modified to incorporate the new constraints. We have developed Vishnu this way, starting with very simple functionality and expanding the capabilities as needed to solve new problems requiring new features. We have yet to encounter a problem that we have not been able to handle by adding new functionality to Vishnu, without affecting existing capabilities.

6 Conclusion

We have developed a powerful framework for representing scheduling problems, and we have built a reconfigurable scheduler, Vishnu, that can find an optimized solution for any problem specified in this framework. The approach we have used involves a genetic algorithm feeding task orderings to a greedy schedule builder as its method of finding optimized schedules. Hooks and formulas provide a method for users to define customized scheduling logic. This approach allows easy introduction of new

capabilities into the scheduler and problem specification framework, thus allowing us to make a reconfigurable scheduler that is particularly powerful in its ability to handle a wide range of scheduling problems with a wide variety of scheduling logic.

The major benefit of Vishnu is that it makes development of optimized scheduling for a wide range of problems simple and inexpensive. There is a vast array of scheduling problems that are currently solved using manual or non-optimized scheduling. For most of these problems, our reconfigurable scheduler could provide a simple and inexpensive optimized scheduling solution.

References

1. Fourer, R., Gay, D., Kernighan, B.: AMPL: A Modeling Language for Mathematical Programming. Duxbury Press, Belmont, CA (1993)
2. Van Hentenryck, P.: The OPL Optimization Programming Language. MIT Press, Cambridge, MA (1999)
3. Montana, D.: A reconfigurable optimizing scheduler. In: Proceedings of the Genetic and Evolutionary Computation Conference. (2001) 1159–1166
4. Montana, D.: Optimized scheduling for the masses. In: Genetic and Evolutionary Computation Conference Workshop Program. (2001) 132–136
5. Bisschop, J., Meeraus, A.: On the development of a general algebraic modeling system in a strategic planning environment. Mathematical Programming Study **20** (1982) 1–29
6. Bixby, R., Fenelon, M., Gu, Z., Rothberg, E., Wunderling, R.: MIP: Theory and practice - closing the gap. In Powell, M., Scholtes, S., eds.: System Modelling and Optimization: Methods, Theory, and Applications. Kluwer (2000) 19–49
7. Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., Berthier, F.: The constraint logic programming language CHIP. In: Proceedings of the International Conference on Fifth Generation Computer Systems. (1988) 693–702
8. Colmerauer, A.: An introduction to Prolog III. Communications of the ACM **28**(4) (1990) 412–418
9. Davis, G., Fox, M.: ODO: A constraint-based architecture for representing and reasoning about scheduling problems. In: Proceedings of the 3rd Industrial Engineering Research Conference. (1994)
10. Van Hentenryck, P., Perron, L., Puget, J.F.: Search and strategies in OPL. ACM Transactions on Computational Logic **1**(2) (2000) 285–320
11. McIlhagga, M.: Solving generic scheduling problems with a distributed genetic algorithm. In: Proceedings of the AISB Workshop on Evolutionary Computing. (1997) 85–90
12. Raggl, A., Slany, W.: A reusable iterative optimization library to solve combinatorial problems with approximate reasoning. International Journal of Approximate Reasoning **19**(1-2) (1998) 161–191
13. Smith, S., Becker, M.: An ontology for constructing scheduling systems. In: Working Notes of 1997 AAAI Symposium on Ontological Engineering. (1997)
14. Rajpathak, D., Motta, E., Roy, R.: A generic task ontology for scheduling applications. In: Proceedings of the International Conference on Artificial Intelligence. (2001) 1037–1043
15. Montana, D.: Introduction to the special issue: Evolutionary algorithms for scheduling. Evolutionary Computation **6**(1) (1998) v–ix
16. Cantu-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Kluwer (2000)

17. Davis, L.: Job shop scheduling with genetic algorithms. In: Proceedings of the First International Conference on Genetic Algorithms. (1985) 136–140

18. Homberger, J., Gehring, H.: Two evolutionary meta-heuristics for the vehicle routing problem with time windows. INFORMS Journal on Computing **37**(3) (1999) 297–318

19. Syswerda, G.: Schedule optimization using genetic algorithms. In Davis, L., ed.: Handbook of Genetic Algorithms. Van Nostrand Reinhold (1991) 332–349

20. Whitley, D., Starkweather, T., Fuquay, D.: Scheduling problems and traveling salesmen: The genetic edge recombination operator. In: Proceedings of the Third International Conference on Genetic Algorithms. (1989) 133–140

21. Goldberg, D., R. Lingle, J.: Alleles, loci, and the traveling salesman problem. In: Proceedings of the First International Conference on Genetic Algorithms. (1985) 154–159

22. Grefenstette, J., Gopal, R., Rosmaita, B., van Gucht, D.: Genetic algorithms for the traveling salesman problem. In: Proceedings of the First International Conference on Genetic Algorithms. (1985) 160–165

23. Giffler, B., Thompson, G.: Algorithms for solving production-scheduling problems. Operations Research **8**(4) (1960) 487–503

24. Beasley, J.: OR-Library: Distributing test problems by electronic mail. Journal of the Operational Research Society **41**(11) (1990) 1069–1072

25. Muth, J., Thompson, G.: Industrial Scheduling. Prentice Hall, Englewood Cliffs, NJ (1963)

26. Solomon, M.: Algorithms for the vehicle routing and scheduling problem with time window constraints. Operations Research **35** (1987) 254–265

27. Montana, D., Brinn, M., Moore, S., Bidwell, G.: Genetic algorithms for complex, real-time scheduling. In: Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics. (1998) 2213–2218

28. Montana, D., Herrero, J., Vidaver, G., Bidwell, G.: A multiagent society for military transporation scheduling. Journal of Scheduling **3**(4) (2000) 225–246

29. Montana, D.: Vishnu reconfigurable scheduler home page (2001) http://vishnu.bbn.com.

30. Fahle, T., Junker, U., Karisch, S., Kohl, N., Sellmann, M., Vaaben, B.: Constraint programming based column generation for crew assignment. Journal of Heuristics **8**(1) (2002) 59–81

31. Hussain, T., Montana, D., Brinn, M., Cerys, D.: Genetic algorithms for UGV navigation, sniper fire localization and unit of action fuel distribution. In: Military and Security Applications of Evolutionary Computation (MSAEC) Workshop, part of GECCO. (2004)

32. Reinelt, G.: TSPLIB (2001)
http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/.

33. Carlier, J., Pinson, E.: Adjustment of heads and tails for the job-shop problem. European Journal of Operations Research **78** (1994) 146–161

34. Applegate, D., Bixby, R., Chvatal, V., Cook, W.: TSP cuts which do not conform to the template paradigm. In Junger, M., Naddef, D., eds.: Computational Combinatorial Optimization. Springer (2001) 261–304

35. Watson, J., Ross, C., Eisele, V., Denton, J., Bins, J., Guerra, C., Whitley, D., Howe, A.: The traveling salesrep problem, edge assembly crossover, and 2-opt. In: Parallel Problem Solving from Nature V. (1998) 823–832