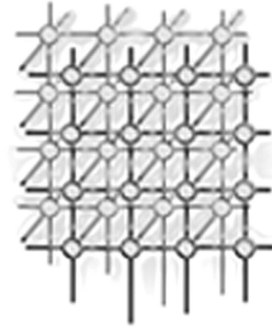


POGGI: Generating Puzzle Instances for Online Games on Grid Infrastructures

Alexandru Iosup*¹

¹ *Parallel and Distributed Systems Group, Delft University of Technology
2628 CD Delft, the Netherlands*



KEY WORDS: game content generation, grid computing, puzzle, MMOG, resource management.

SUMMARY

The developers of Massively Multiplayer Online Games (MMOGs) rely on attractive content such as logical challenges (puzzles) to entertain and generate revenue from millions of concurrent players. While a large part of the content is currently generated by hand, the exponentially increasing number of players and their new demand for player-customized content have made manual content generation undesirable. Thus, in this work we investigate the problem of automated, player-customized game content generation at MMOG scale; to make our investigation tractable, we focus on puzzle game content generation. In this context, we design POGGI, an architecture for generating player-customized game content using on-demand, grid resources. POGGI focuses on the large-scale generation of puzzle instances that match well the solving ability of the players, and that lead to fresh playing experience. Using our reference implementation of POGGI, we show through experiments in a real resource pool of over 1,600 nodes that POGGI can generate commercial-quality content at MMOG scale.

1. Introduction

Massively Multiplayer Online Games (MMOGs) have emerged in the past decade as a new type of large-scale distributed application: real-time virtual world simulations entertaining at the same time millions of players located around the world. In real deployments, the operation and maintenance of these applications includes two main components, one running the large-scale virtual world simulation, the other populating the virtual world with content that keeps the players engaged (and paying). So far, content has been provided exclusively by human content designers, but the growth of the player population, the lack of scalability of the production pipeline, and the increase in the price ratio between human work and computation make this situation undesirable for the future. Extending our previous work [14], here we investigate the problem of automated, player-customized content generation for MMOGs using grids.

*Correspondence to: Email: a.iosup@tudelft.nl. We thank Dr. Dick Epema and Dr. Miron Livny for support.



There are many types of content present in MMOGs, from 3D objects populating the virtual worlds, to abstract challenges facing the players. Since content generation is time-consuming and costly, MMOG developers seek the content types that entertain and challenge players for the longest time. *Puzzle games*, that is, games in which the player is entertained by solving a logical challenge, are a much sought-after MMOG content type; for instance, players may spend hours on a chess puzzle [17, 24] that enables exiting a labyrinth. Today, puzzle game content is generated by teams of human designers, which poses scalability problems to the production pipeline. While several commercial games have made use of content generation [1, 5, 7, 29], they have all been small-scale games in terms of number of players, and did not consider the generation of puzzle game content. In contrast to this body of work, we have started [14] to investigate the large-scale generation of puzzle game instances for MMOGs on grid infrastructures. Our initial contribution has been threefold: the formulation of the problem of puzzle game generation as a novel large-scale, compute-intensive application different from typical applications from other applications present in grids and parallel production environments; the introduction of an application model and an architecture, POGGI, for generating puzzle games at large scale; and an experimental evaluation of the proposed architecture. The main contribution in this work is as follows:

1. We reformulate the problem of puzzle game generation to include the requirements of player-customized content (Section 2);
2. After revisiting the application model (Section 3), we extend the POGGI architecture with player-customized content generation at large scale (Section 4);
3. We show through experiments in a real environment that our proposed architecture can generate at large scale content of commercial quality (Section 5).

2. Problem Formulation

In this section we formulate the problem of generating puzzle content for MMOGs.

2.1. The Need for Automatic Puzzle Content Generation

From the many existing game genres [34], the puzzles genre is one of the most attractive for MMOG content designers. First, puzzles are preferred over other genres by a majority of the online game players [34, p.24]. Second, puzzles can be used to entertain the players who prefer thinking over repetitive activities, either by embedding in MMOGs such as World of Warcraft or by building with them themed MMOGs such as Yahoo! Games. Third, puzzles can be used as building blocks for many types of games, from chess puzzles to classic adventures such as Lucas Arts's Day of the Tentacle and Grim Fandango [30]. Fourth, and in contrast to other content, the difficulty and the attractiveness of a puzzle instance can easily be quantified by metrics such as "the number of steps for the best solution" and "number of twists in the path".

In this work we show that puzzles have a fifth attractive feature: customized puzzle instances that are attractive and challenging can be automatically generated at the massive scale required by MMOGs.

2.2. Challenges in Puzzle Game Content Generation for MMOGs

We identify three main challenges in solving *the problem of generating customized puzzle instances for MMOGs*: puzzle difficulty balance, puzzle freshness, and scalability to large numbers of players. Failing to meet any of these challenges has a direct impact on revenues:

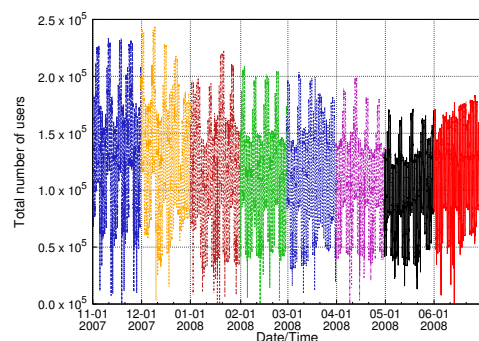


it leads to a degraded game experience and causes the players to leave the game [3, 12]. We explain each of the three challenges in turn.

Many MMOG players get entertained by gaining virtual reputation with their ability to excel in game activities such as puzzle solving [3, ch.3]. To keep the reputation system fair, the puzzle instances given to a player to solve must be matched with the ability of the player to solve it, and be comparable in difficulty for players of the same ability; we call this combined challenge the *puzzle difficulty balance*.

Similarly, many (other) MMOG players enjoy the feeling of being “first” [3, ch.3]: they want to explore new territory, to experience new challenges, to solve a new puzzle. Thus, the main challenge is to give each player to solve puzzle instances that lead to a fresh experience, both visual and auditive; we call this challenge *puzzle freshness*.

The ability to scale to the large numbers of players specific to MMOGs means



The number of Active Concurrent Players in the RuneScape MMOG over time. Each month of data is depicted with a different type of line.

Figure 1.

sampling interval. Figure 1 shows that for RuneScape the daily ACP peaks are between 100,000 and 250,000 players world-wide. (Peaks are several hours long.)

2.3. Current Industry Approach

Due to the business model associated with MMOGs, in which the content is the main factor in attracting and retaining paying players [3], game operators prefer with few exceptions to generate the content in-house. This is currently achieved by employing for a large MMOG tens of (human) content designers; of these, a significant number form the management hierarchy, raising the production cost without directly contributing to content generation. As a result, manual puzzle instance generation does not scale to larger output, due to cost and time overheads, and in particular it will not scale with the growing MMOG-playing community, which over the last decade exhibits an superlinear (exponential) increase in size [20, 37]; even for popular MMOGs such as World of Warcraft and Runescape, the number of quest and puzzle instances is in the order of hundreds to thousands, whereas the population is in the order of millions. This scalability problem is made more difficult by the impossibility to know



	Workload Type		
	Puzzle Gen.	Grids [15]	Parallel Production Environment [18]
Users	100,000s	10s-100s	10s-100s
Workload	Very Dynamic	Dynamic	Static
Response Time	Very Low	High	Very High
# of Jobs/Result	10s-1,000s	10s-100,000s	1-100s
Task coupling	Workflows	Workflows, Bags-of-Tasks	Parallel, Single

Table I. Comparison between Puzzle Generation and traditional high-performance workloads.

in advance how the player population will evolve over time: for some MMOGs the population will ramp up or scale down dramatically in a matter of weeks or even days [20].

Faced with scalability problems, the dominant industry practice is to generate mostly low-difficulty puzzles, and in such quantity that only a small fraction of the players can enjoy “fresh content”; we have shown in previous work [13] that this may not satisfy the requirements of today’s MMOG population, which contains many players that have stayed in the system long enough to have reached medium to high skill levels. Another downside of this practice is the proliferation of cheating: the puzzles and quests, which are present in limited amount in each MMOG, are quickly solved by top players who, in turn, create or contribute to online lists of known instances/solutions that players of lower levels can exploit; for RuneScape, sites such as Zybez.net and RuneHQ.com publish solutions to puzzles and quests only days after they become playable within the game. Thus, few players ever get to solve a new puzzle or quest, or to solve puzzle instances that lead to unique visual and aural experience.

An alternative to generating content in-house is to use the machines or even the creativity and time of the game players to generate content. However attractive this approach may seem, its adoption requires good answers for three main research questions. First, *How to control production pipeline?* Game developers sell content not technology, and user-generated content threatens the developer’s main reason for existing. Second, *How to avoid game exploits?* A player may cheat by producing difficult puzzle instances which only few players can solve but that have a hidden “key”; this may lead to unfair virtual and financial gains to the cheater [21]. Third, *How to select the most interesting parts from the content submitted by users?*

2.4. Our Vision: Puzzle Content Generation using Grid Infrastructures

Our vision is that the puzzle game content will be automatically generated by a computing environment that can dynamically adjust to a large-scale, compute-intensive, and highly variable workload. Grid environments and to some extent even today’s parallel production environments (PPEs) match well the description of this environment. However, whether these environments can support the MMOG puzzle generation workloads and their associated challenges is an open research question. Table I compares the characteristics of the MMOG workloads with those of the typical workloads present in these environments. Both the current grids and PPEs serve two-three orders of magnitude fewer users than needed for MMOGs. Grids and PPEs handle well dynamic and static workloads, but have problems running very dynamic workloads [15, 25, 32] such as MMOGs’ (which feature many bursts of jobs and much larger workflows, see Section 5.2). The response time model for grid and PPE applications permits middleware overheads higher than what is acceptable for MMOG workloads. Efficiently running in grids large workflows or large numbers of related jobs to produce a unique final result are still active research topics [25].



2.5. The Lunar Lockout Scenario

We conclude our problem formulation with a motivating scenario: the automatic generation of instances for Lunar Lockout [38], a commercial puzzle designed by Hiroshi Yamamoto and Nob Yoshigahara. This puzzle continues to be sold unchanged in the medium-difficult puzzles market segment; thus, our focus on it is akin to the focus of the AI community on traditional but difficult games such as go, chess, and bridge.

Lunar Lockout consists of a square board on which several pins are placed before the puzzle solving begins. Pins must be moved according to the game rules such that one marked pin reaches a target cell on the board. We present only informally the game rules; a formal set of rules and an analysis of the number of unique instances are also available [27]. A move consists of pushing continuously a pin either horizontally or vertically, until it is blocked by another pin. The moved pin will be placed in the board cell just before the cell of the blocking pin, in the direction of the push. The moved pin cannot hop over another pin, or be moved or end the move outside the board. The goal is to place the X pin on the target cell. A *solution* consists of a time-ordered set of pin movements leaving the X pin to the target. The puzzle is characterized by the size of the board N , the number of pins P , and the difficulty settings D (i.e., number of moves for an entertaining solution depending on the player's level). A *puzzle instance* consists for Lunar Lockout of three pieces of information: N , P , and the initial position of the pins; see Section 5.1 for a generated puzzle instance, and Figure ?? for a generated puzzle instance and its optimal solution.

Only a few tens of Lunar Lockout puzzle instances have been released to the puzzle community since the initial presentation of the puzzle, and the commercial variants of this game total fewer than one hundred puzzle instances. However, an analysis of the number of instances for a board size $N = 5$ and number of pins $P \in [2, 7]$ shows that there exist millions of interesting instances [27]; the number of instances greatly increases with larger N and/or P . We consider generating Lunar Lockout instances at massive scale as a motivating scenario for our work. Similar puzzles have been included in leading MMOGs such as World of Warcraft and Runescape; the instances present in these MMOGs are small and repetitive.

3. Application Model

In this section we model puzzle content generation as a workflow; ours is the first workflow formulation for this novel application domain.

The puzzle instance generation application consists of two main functional phases: generating a solvable puzzle game instance and finding a solution for it, and testing that the proposed solution is minimal in solving effort. The second phase addresses the puzzle difficulty balance challenge (see Section 2.2), so that players cannot solve a generated puzzle in a simpler way. The two functional phases are executed until both complete successfully, or until the time allocated for their execution is exceeded. Thus, our puzzle generation application is one of the first iterative grid workflows; given the number of iterations observed in practice (see Section 5.2), it is also one of the largest.

3.1. Workflow Structure

We model the generic puzzle game generation application as the workflow with seven levels depicted in Figure 2: *Generate Setup*, *Find and Verify Solution*, *if Solution*, *Test Solution*



Optimality/Suitability, if Passed Test, Record Setup and Solution, and if Solution Meets Difficulty Settings. The first three (the following two) levels correspond to the first (second) functional phase of the workflow; the other levels are added for output and ending purposes.

The *Generate Setup* level generates a puzzle game setup, such as a board and the pieces on it, that needs to be solved. The execution time of this level depends on what functionality it provides: this level may be designed to generate random setups including ones that break the rules of the puzzles, or include the verification of the setup validity against the rules.

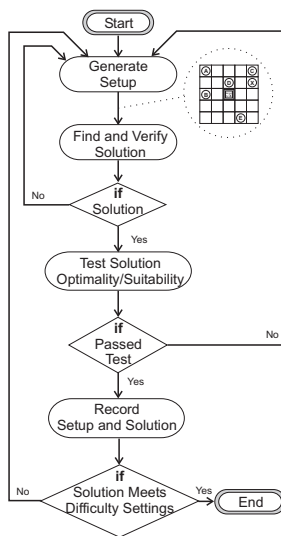


Figure 2. The workflow model.

The *Find and Verify Solution* level solves the puzzle starting from the setup generated in the previous level and using one of the many game solving techniques to evolve from setup to the solution state; for reviews of these techniques we refer to [6,8,9]. Depending on the amenability of the puzzle to different solving strategies, this level may employ algorithms from brute force search to genetic algorithms, game-tree search, and SAT-solving. Regardless of the algorithm employed, the execution time for this level is bounded by the maximum execution time set by the application designer.

The *Test Solution Optimality / Suitability* level attempts to determine if the found solution is optimal, and if it is suitable for being recorded (i.e., is the solution of the right complexity, but also of the type that will entertain the players?). Depending on the puzzle and on the setup/solving algorithms chosen by the developer, the *Test Solution Optimality / Suitability* level may require similar or even more computing cycles than the *Find and Verify Solution* level—for example, if the puzzle movements cannot lead to a previously explored state, an initial setup may be generated by moving the pieces from a starting position in reverse, akin to retrograde analysis [28]; then, finding the solution requires only reversing a list, but the Test level will need to solve the game and test the optimality of the solution.

The *if Solution Meets Difficulty Settings* level was introduced at the end of the workflow to ensure the recording of puzzle setups whose best solutions do not meet the current difficulty settings, but have passed all the other tests and may therefore be used for lower difficulty settings.

3.2. Example: the Lunar Lockout Puzzle

We demonstrate the use of the workflow-based application model by applying it to the Lunar Lockout scenario introduced in Section 2.5.

The *Generate Setup* level generates a random positioning of the pins on the board, ensuring that the basic rules of the game are enforced, e.g., the pins do not overlap.

The *Find and Verify Solution* level uses backtracking to solve the puzzle. This corresponds to the situation common in practice where no faster solving algorithms are known. The choice of backtracking also ensures low memory consumption and efficient cache usage for today's processors. This workflow level stops when a solution was found.



The *Test Solution Optimality / Suitability* level also applies backtracking to find all the other solutions with a lower or equal number of movements (size). This level needs to investigate moves that were not checked by the *Find and Verify Solution* level; for backtracking this can be easily achieved with minimal memory consumption and without redundant computation. If a solution with a lower number of movements is found, it becomes the new solution and the process continues.

4. The POGGI Architecture

In this section we present the Puzzle-Based Online Games on Grid Infrastructures (POGGI) architecture for puzzle game generation on grid infrastructures.

4.1. Overview

The main goal of POGGI is to meet the challenges introduced in Section 2.2. In particular, POGGI is designed to generate puzzle game content efficiently and at the scale required by MMOGs by executing large numbers of puzzle generation workflows on remote resources. We focus on three main issues:

1. *Reduce execution overheads* First, by not throwing away generated content that does not meet the current but may meet future difficulty settings (see the last workflow level in Section 3.1) our architecture efficiently produces in advance content for future needs. Second, in previous work [32] we have found the job execution overhead to be an important performance bottleneck of current grid workflow engines; for this reason, no current (grid) workflow engine can be used to handle the large number of tasks required by our application (see Section 5.2). To address this problem, we have developed a workflow engine dedicated to puzzle content generation.

2. *Adapt to workload variability* Using the long-term traces we have acquired from one of the Top-5 MMOGs, we have recently shown [20] that MMOGs exhibit high resource demand variability driven by a user presence pattern that changes with the season and is also fashion-driven, and by a user interaction pattern that changes with the gameplay style. We use detailed statistics of the puzzle use and of the content generation performance to adapt to workload variability.

3. *Use existing middleware* The architecture uses an external Resource Management Service (RMS), e.g., Condor [33], Globus and Falcon [26], to execute reliably bags-of-tasks in which each task is a puzzle generation workflow, and to monitor the remote execution environment. The architecture is also interfacing with an external component, *Game Content*, which stores the generated content, and which collects statistical data about the actual usage of this content.

4.2. Main Components

The six main components of POGGI are depicted in Figure 3, with rectangles representing (nested) components, and arrows depicting control and data flows.

1. *Workflow Execution Engine for Puzzle Game Generation* and 2. *Post-Processing*: Generic workflow execution engines that can run the puzzle generation workflows (see Section 3.1) already exist [19, 22, 31, 36]. However, they are built to run the workflow tasks on remote resources through the services of the RMS, which leads to high communication and state management overheads [26, 32]. In contrast to this approach, we introduce in our architecture a specialized component for the execution of the puzzle generation workflows on a single

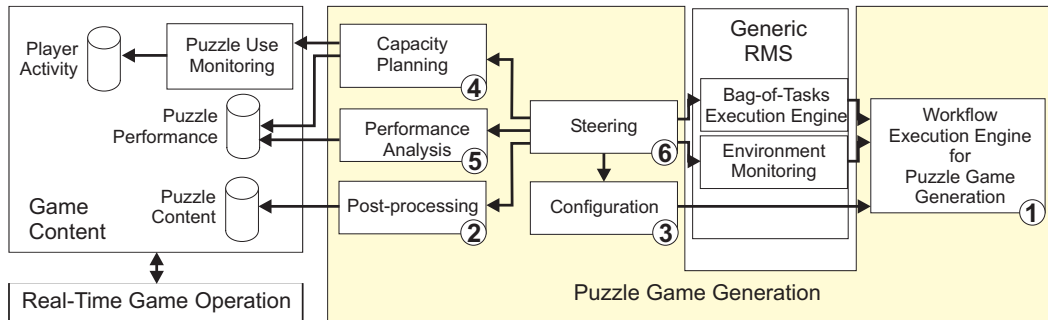


Figure 3. Overview of the POGGI architecture for puzzle game content generation.

resource; we show in Section 5 that our workflow engine can handle the large number of tasks required for puzzle content generation. While this component can be extended to execute workflow tasks on multiple resources similarly to the execution of bag-of-tasks by Falkon [26], the presence of thousands of other workflow instances already leads to high parallelism with minimal execution complexity. The *Post-Processing* component parses the workflow output and stores the generated content into a database.

3. Configuration: The generic workflow model introduced in Section 3.1 can generate content for all the difficulty levels, if the difficulty settings are set to the maximum and the suitability level is set to the minimum. Then, all the generated setups that are solvable are recorded. However, this configuration is inefficient in that high difficulty states will be explored even for beginner players. Conversely, settings that are too strict may lead to ignoring many possibly useful setups. Thus, we introduce in the architecture a component to set the appropriate configuration depending on the level of the players in need of new content. We compare in Section 5.3 two configurations; exploring configuration policies is outside the focus of this work.

4. Capacity Planning: As shown in Figure 1, the ACP volume reaches daily peaks of twice the long-term average. In addition, the use of a specific puzzle game may be subject to seasonal and even shorter-term changes. Thus, it would be inefficient to generate content at constant rate. Instead, the capacity planning component analyzes the use of puzzles and the performance of the content generation process and gives recommendations of the needed number of resources. Given the high number of players, enough performance analysis data is present almost from the system start-up in the historical records. We have evaluated various on-line prediction algorithms for MMOG capacity planning in our previous work [20].

5. Performance Analysis: Detailed performance analysis is required to enable the capacity planning component. The performance analysis component focuses on two important performance analysis aspects: extracting metrics at different levels, and reporting more than just the basic statistics. We consider for performance analysis job-, operational-, application-, and service-level metrics. The first two levels comprise the traditional metrics that describe the execution of individual [11] and workflow [35] jobs. The *application-level metrics* and the *service-level* metrics are specific to the puzzle generation application. At the application-level the analysis follows the evolution of internal application counters (e.g., number of states explored) over time. At the service-level the analysis follows the generation of interesting puzzle

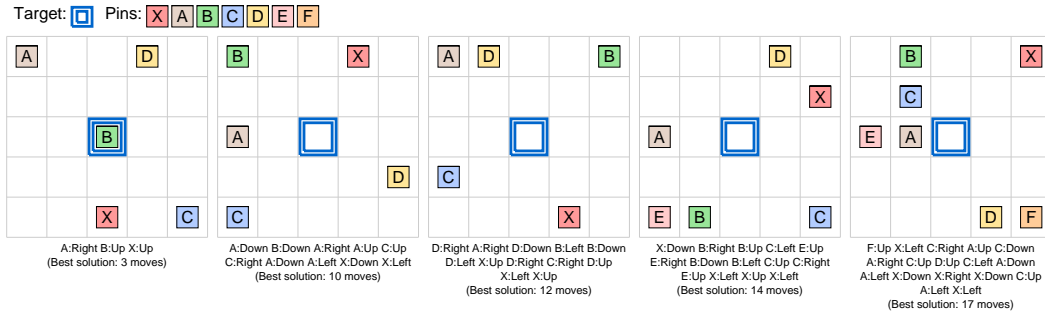


Figure 4. Examples of generated puzzle instances solution sizes from 3 to 17 moves ($N = 5$).

game instances. The performance analysis component performs an in-depth statistical analysis of metrics at all levels.

6. *Steering*: To generate puzzle game instances, the various components of our architecture need to operate in concert. The steering component triggers the execution of each other component, and forcefully terminates the puzzle generation workflows that exceed their allotted execution time. Based on the capacity planning recommendations and using the services of the generic RMS, it executes the puzzle generation workflows.

4.3. Puzzle difficulty vs. solving ability

Two problems arise when meeting the challenge of puzzle difficulty balance: evaluating the difficulty of puzzle instances and evaluating the solving ability of the players. We describe in the following our solution for each problem, in turn.

4.3.1. Evaluating the difficulty of puzzle instances

We consider two types of metrics for assessing puzzle instance difficulty: puzzle-agnostic and puzzle-specific. The puzzle-agnostic metrics include all the metrics that can be characterize the difficulty of any puzzle, such as “the number of moves in the optimal solution” (*solution size*) and “the number of possible solutions” (*alternatives*), applied to a puzzle instance. In contrast, puzzle-specific metrics are tightly coupled to the puzzle for which they were designed, and thus cannot be applied directly to another puzzle. It is our conjecture that puzzle designers can find expressive yet puzzle-agnostic metrics, which can then be applied for a wide variety of puzzles; we will explore this conjecture in future work. To use these metrics in combination, the puzzle designer can assign weights to each metric; validating the weights for a specific puzzle is outside the scope of this article.

We consider for Lunar Lockout the puzzle-agnostic metrics *solution size* and *alternatives*. We also consider two puzzle-specific metrics that reflect how a player may perceive the instance. The first is “Pin-on-Target”, an instance attribute defined only when a pin (other than X) is initially placed on the target position. Compared with other instances of equal solution size, for “Pin-on-Target” at least one move is suggested from start: relocating the pin from the target position. The second Lunar Lockout-specific metric that we consider is the *skill move*, that is, a group of consecutive moves that involve the same pin. The skill move concept is generic: skill moves of length 3 are often employed in Grim Fandango, e.g., the move to location-get object-mix objects sequence is used twice in the Diffuse Bomb puzzle [30, p.58]. From the puzzle



instances depicted in Figure 4, the leftmost is of type “Pin-on-Target” with solution size of 3 moves, and the rightmost has a solution which includes a skill move of length 3 (pin X).

4.3.2. Evaluating the solving ability of the players

We design a history-based mechanism for evaluating the solving ability of the players. We keep for each player a record of the tried and of the solved puzzles instances; the set of all records is the historical database. From the historical database it is easy to extract the current level of a player, which is the level just above the maximum difficulty level of a puzzle solved by this player in the past. For Lunar Lockout, the current level of a player is equal to the maximum level of a solved instance plus one. The historical database also provides information about the complete player population. We consider two pieces of information: the distribution of solving ability (*skill level*), and the time it takes for the solving ability to increase (*leveling time* [10]). When generating new instances for a player their difficulty settings can be set automatically, based on the player’s current level, on the real skill level and leveling time of the player population, and on the game designer’s target skill level and leveling time.

As an example of skill level for a player population, we have analyzed the skill level of the players population of RuneScape, the second-most popular MMOG today [20]. To this end, we have developed CAMEO [13], a data collection platform that provisions its resources dynamically. Using CAMEO, we have collected official skill level data for over 2,500,000 players, and shown [13] that the number of high-level players is significant and that the most populated skill levels are those corresponding to the middle skill level. This further demonstrates the importance of POGGI: the current approach of generating mostly low-difficulty content does not suit the real player capabilities of modern MMOGs.

4.4. Generating fresh puzzles

In contrast to the current industry approach (see Section 2.3), our approach is to generate different puzzle game instances for each player (while adhering to the puzzle difficulty balance), and to give each player to solve puzzle instances that lead to different visual and aural experience. Thus, we facilitate the emergence of a “uniqueness” feeling for each player. As a positive consequence, this also reduces the usefulness of online lists of known instances/solutions, thus reducing cheating.

Similarly to our approach for evaluating the difficulty of puzzle instances, we consider two types of methods to generate fresh puzzles for each player. For puzzle-agnostic generation we employ random walks, that is, a generic method for randomly exploring the very large space of potential puzzle instances. Many of these random instances cannot be used: they are unsolvable, or they have too short or too large solution sizes. For Lunar Lockout it turns out [27] that if one instance is solvable, by changing from this instance the position of a single pin in many cases the new instance is also solvable, though with very different difficulty properties.

We also consider a set of three puzzle-specific rules to increase the freshness of generated puzzles; other rules can be easily devised. First, we assume that each pin has a different animation when moved; then, we consider solutions to be fresh if they involve most or all of the pins, thus displaying all the animations. When an instance has many solution alternatives, the creative designer can push the player to find fresher solutions by allocating different resources for each pin, and by limiting the amount of resources. Second, we assume that pins have different animations for movement in different directions, and we select for a player boards



that will involve different types of moves. Third, we observe that the designer can also use the puzzle-specific difficulty metrics introduced in Section 4.3.1, such as the skill move—these metrics translate well into puzzle-specific rules to increase the freshness of generated puzzle instances.

5. Experimental Results

In this section we present our experimental results, which demonstrate that POGGI can be used in real conditions to generate commercial-quality content.

We have implemented the Steering component of our architecture on top of the GrenchMark [16, 32] grid testing tool, which can already generate and submit multi-job workloads to common grid and cluster resource management middleware such as Condor, Globus, SGE, and PBS. Thus, the POGGI architecture is not limited to a single middleware, and can already operate in many deployed environments. For this work we have extended the workload generation and management features of GRENCHMARK, in particular with the ability to generate the bags-of-tasks comprising puzzle generation workflows. We have built the Performance Analysis component on top of the GrenchMark tool for analyzing workload execution, which can already extract performance metrics at job and operational levels [32]. We have added to this component the ability to extract performance metrics at the application and service levels.

We have performed the experiments in the Condor pool at U.Wisconsin-Madison, which comprises over 1,600 processors. The system was shared with other users; for all experiments we have used a normal priority account.

5.1. Lunar Lockout: Solved and Extended

The commercial version of Lunar Lockout [38] consists of a playing set and cards describing 40 puzzle instances. The instances have been generated manually by a team of three content designers over a period of about one year. The instances can be characterized in our application model (see Section 3.2) as $N = 5$, $P = 4 - 6$, and D such that the solution size ranges from 4 (beginner player) to 10 (advanced). During our experiments we have generated and solved all the boards that come with the commercial version of the game. This demonstrates that our architecture can be used to produce commercial-quality content much faster than by using the manual approach.

The puzzle instances included in the commercial version of Lunar Lockout are of low (solution size 4-7) and moderate (solution size 8-10) difficulty, but only three of the forty instances are of moderate difficulty. We have emphasized in Section 4.3.2 that for a well-designed MMOG the most common skill level is moderate and, more importantly, that there exist many players of high skill levels. Using our approach, we have generated automatically many new puzzle instances with equal or larger boards ($N \geq 5$), more pins ($P > 6$), and solutions of up to 21 moves, corresponding to an expert play level that exceeds the human content design capabilities; Figure 4 depicts several auto-generated puzzle instances. Sizes of more than 15 moves are very difficult, and correspond to the largest commercial (multi-

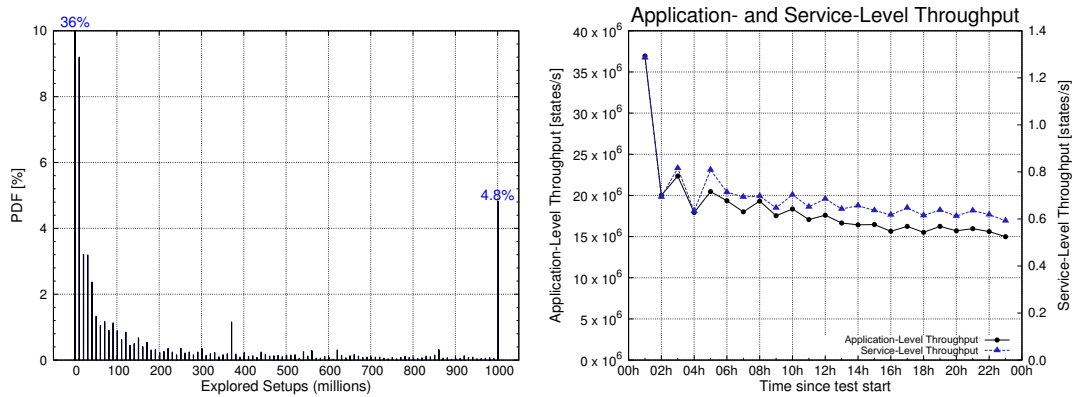


Figure 5. (left) The PDF of the number of explored puzzle setups per workflow. Each bar represents a range of 10 million puzzle setups. (right) The application- and service-level throughput over time.

)puzzles: for Grim Fandango [30][†] the *Chase Hector*, *Get Access to Sprouter*, and *Make Vacancy* puzzles have been designed to have solutions of size 21, 20, and 18, respectively. This shows that POGGI is able to produce puzzle instances whose difficulty matches and even exceeds that of complex commercial puzzles.

5.2. Application Characterization

To characterize the puzzle generation application we use a test workload comprising 10,000 workflows. Overall, the execution of this workload led to the evaluation of 1,364 billion of puzzle setups (tasks). The number of tasks is much larger for puzzle content generation than for the largest scientific workflows; the latter comprise rarely over a million of tasks [4, 23, 25].

The probability distribution function (PDF) of the number of explored puzzle setups for this workload is depicted in Figure 5 (left). The distribution is skewed towards left: most of the executed workflows explore fewer puzzle setups than the average of the whole workload. This indicates that in most cases the workflow termination condition (finding a puzzle setup that matches the difficulty settings) is met faster than the workload average indicates.

5.3. Meeting the Challenges

We have identified in Section 2.2 three main challenges for puzzle content generation: puzzle difficulty, puzzle freshness, and scalability. We now show evidence that the POGGI architecture can meet these challenges.

To show evidence of meeting the challenge of puzzle difficulty balance we have generated puzzles for a player of increasing ability; Figure 4 shows a selection from the generated instances. From the puzzle-agnostic point of view, all puzzle instances are on boards of size $N = 5$, and the number of moves for an optimal solution increases from 3 to 17. The number

[†]Grim Fandango signaled the end of multi-puzzle game and the growth of the single puzzle market. While it won prestigious awards after its release in 1998—PC Gamer’s Adventure Game of the Year, IGN’s Best Adventure Game of the Year, and GameSpot’s PC Game of the Year—, it only sold well, not very well.



Configuration	Solution Size																
	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Normal	1,281	1,286	645	375	230	134	75	47	27	11	6	1	2	-	-	-	-
Large	-	-	409	334	257	147	171	57	79	83	39	41	24	22	2	3	7

Table II. The number of interesting states found for each 1,000 jobs, per solution size.

of pins also increases, from $P = 5$ for solution sizes 3 to 12 to $P = 7$ for solution sizes 15 to 17. Thus, the player meets with increasingly more challenging problems. From the puzzle-specific point of view, the first of the puzzle instances with the highest number of pins, the instance with solution size 14, includes several skill moves (e.g., $B \rightarrow \text{Right}$, Up). Thus, the player is gradually introduced to the most difficult instances. There are several puzzle instances with a pin starting on the target position, such as the puzzle instances with solution size of 3. Also, in several puzzles at least one pin does not need to be moved at all, such as the pins C and D in the puzzle instance with solution size of 3. This allows the beginning player to learn the game and to adapt to increasing puzzle difficulty.

We evaluate the impact of the application configuration on finding puzzle instances of different difficulty. Two configurations are considered during this experiment: workflows that explore a normal-sized space (*normal instances*), and workflows that explore a large-sized space (*large instances*). Table II shows the number of puzzle instances found for these two configurations. The normal workflow instances find more puzzle instances with solution sizes up to 7. For solution sizes of 8 through 12, both instances behave similarly. For solution sizes of 13 and higher, the large workflow instances become the best choice. Based on similar results and on demand the Capacity Planning component can issue recommendations for efficiently finding unique instances of desired difficulty.

To show evidence of scalability we investigate the average response time and the potential for soft performance guarantees. For the 10,000 workflows test workload described in Section 5.2, the average workflow is computed in around 5 minutes; thus, it is possible to generate content for hundreds of thousands of players on a moderately sized grid infrastructure.

We have also evaluated the application- and the service-level throughput over time. We define the application-level (service-level) throughput as the number of (interesting) puzzle setups explored (found) over the time unit, set here to one second; for convenience, we use the terms states and puzzle setups interchangeably. Figure 5 (right) shows the evolution of application- and service-level throughput over time. The architecture achieves an overall application-level throughput of over 15 million states explored per second, and an overall service-level throughput of over 0.5 interesting states discovered per second. The performance decreases with time due to Condor's fair sharing policy: our normal user's priority degrades with the increase of resource consumption. The performance decrease becomes predictable after about 6 hours. This allows a service provider to practically guarantee service levels, even in a shared environment. These experiments also demonstrate the ability of our architecture to extract application- and service-level metrics.

6. Related Work

In this section we survey two areas related to our work: game content generation, and many-tasks applications. We have already presented in Section 2.3 the current industry practice.



The topic of automated (procedural) generation has already been approached by the industry as an alternative to manual content generation. In 1984, the single-player game *Elite* [7] used automated game content generation to simulate a large world on an 8-bit architecture. Since then, several other games have used a similar approach: *ADOM* [5] generates battle scenes that adapt dynamically to the player level, *Diablo* [29] generates instances of enclosed areas for the player to explore, *The Dwarf Fortress* [1] generates an entire world from scratch, etc. All these approaches were games with a small numbers of concurrent players or even with a single player, and generated content on the (main) player's machine. In contrast, this work focuses on the efficient generation of puzzle game instances for MMOGs on a resource pool.

Five years ago, few environments existed that could manage the high number of jobs required by MMOGs, among them SETI@Home [2]. More recently, tools such as *Falkon* [26] and *Swift* (through *Falkon*) have started to address the problem of executing with low overhead large numbers of bags-of-tasks and workflows, respectively. In contrast with these approaches, our architecture optimizes the execution of a specific application (though with a much wider audience) and specifically considers dynamic resource provisioning adjustments to maintain the performance metrics required by application's commercial focus.

7. Conclusion and Ongoing Work

With a large and growing user base that generates large revenues but also raises numerous technological problems, MMOGs have recently started to attract the interest of the research community. In this work we focus on the problem of the scalability of player-customized content generation. To address this problem, we have designed an implemented POGGI, an architecture for automatic and dynamic puzzle instance generation at MMOG scale. Experimental results in a large resource pool show that our approach can achieve and even exceed the manual generation of commercial content.

Currently, we are extending our architecture with more puzzle types, and with execution on top of cloud computing resources.

REFERENCES

1. Tarn Adams. *Dwarf Fortress*. Free. [Online] Available: <http://www.bay12games.com/dwarves/>, 2009.
2. David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@Home. *Commun. ACM*, 45(11):56–61, 2002.
3. Richard Bartle. *Designing Virtual Worlds*. New Riders Games, 2003. ISBN 0131018167.
4. Shishir Bharathi, Ann Chervenak, Ewa Deelman, Gaurang Mehta, Mei-Hui Su, and Karan Vahi. Characterization of scientific workflows. In *Workshop on Workflows in Support of Large-Scale Science (WORKS08)*, pages 1–11, 2008.
5. Thomas Biskup. *ADOM*. Free, 1994. [Online] Available: <http://www.adom.de/>, 2009.
6. Bruno Bouzy and Tristan Cazenave. Computer go: An ai oriented survey. *Artificial Intelligence*, 132:39–103, 2001.
7. David Braben and Ian Bell. *Elite*. Acornsoft, 1984. [Online] Available: <http://www.iancgbell.clara.net/elite/>, 2009.
8. J. H. Conway. All games bright and beautiful. *The American Mathematical Monthly*, 84(6):417–434, 1977.
9. Erik D. Demaine. Playing games with algorithms: Algorithmic combinatorial game theory. In *Proc. Symp. on Math Found. in Comp. Sci.*, LNCS, pages 18–32. Springer-Verlag, 2001.
10. Nicolas Ducheneaut, Nick Yee, Eric Nickell, and Robert J. Moore. Building an MMO with mass appeal. *Games and Culture*, 1(4):281–317, Oct 2006.



11. Dror G. Feitelson and Larry Rudolph. Metrics and benchmarking for parallel job scheduling. In *IPPS/SPDP*, volume 1459 of *LNCS*, pages 1–24. Springer, 1998.
12. Tobias Fritsch, Hartmut Ritter, and Jochen H. Schiller. The effect of latency and network limitations on mmorpgs: a field study of everquest2. In *NETGAMES*. ACM, 2005.
13. Alexandru Iosup. Cameo: Continuous analytics for massively multiplayer online games. In *ROIA 2009*, *LNCS*, pages 1–10. Springer, 2009. in conjunction with Euro-Par 2009. (in print).
14. Alexandru Iosup. Poggi: Puzzle-based online games on grid infrastructures. In *Euro-Par*, volume 5704 of *LNCS*, pages 390–403, 2009.
15. Alexandru Iosup, Catalin Dumitrescu, Dick H. J. Epema, Hui Li, and Lex Wolters. How are real grids used? the analysis of four grid traces and its implications. In *GRID*, pages 262–269. IEEE, 2006.
16. Alexandru Iosup and Dick H. J. Epema. GrenchMark: A framework for analyzing, testing, and comparing grids. In *CCGrid*, pages 313–320. IEEE, 2006.
17. Gary Kasparov. *Chess Puzzles Book*. Everyman Chess, 2001.
18. Uri Lublin and Dror G. Feitelson. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *J.PDC*, 63(11):1105–22, 2003.
19. Ludäscher, B. et al. Scientific workflow management and the Kepler system. *Conc.&Comp.: Pract.&Exp.*, 18(10):1039–1065, 2006.
20. Vlad Nae, Alexandru Iosup, Stefan Podlipnig, Radu Prodan, Dick H. J. Epema, and Thomas Fahringer. Efficient management of data center resources for massively multiplayer online games. In *ACM/IEEE SuperComputing*, 2008.
21. Christoph Neumann, Nicolas Prigent, Matteo Varvello, and Kyoungwon Suh. Challenges in peer-to-peer gaming. *Computer Communication Review*, 37(1):79–82, 2007.
22. Oinn, T. M. et al. Taverna: lessons in creating a workflow environment for the life sciences. *Conc.&Comp.: Pract.&Exp.*, 18(10):1067–1100, 2006.
23. S. Ostermann, A. Iosup, R. Prodan, T. Fahringer, and D.H.J. Epema. On the characteristics of grid workflows. In Sergei Gorlatch, editor, *Proc. of the CoreGRID Workshop on Integrated Research in Grid Computing (CGIW'08)*, pages 431–442. CoreGRID, Apr 2008.
24. Laszlo Polgar. *Chess: 5334 Problems, Combinations and Games*. Leventhal, 2006.
25. Ioan Raicu, Zhao Zhang, Mike Wilde, Ian Foster, Pete Beckman, Kamil Iskra, and Ben Clifford. Toward loosely-coupled programming on petascale systems. In *ACM/IEEE SuperComputing*, 2008.
26. Ioan Raicu, Yong Zhao, Catalin Dumitrescu, Ian T. Foster, and Michael Wilde. Falcon: a fast and light-weight task execution framework. In *ACM/IEEE SuperComputing*, 2007.
27. John Rausch. Computer analysis of the ufo puzzle. Online Report, 2001. [Online] Available: <http://www.johnrausch.com/PuzzleWorld/art/art03.htm>.
28. John W. Romein and Henri E. Bal. Solving awari with parallel retrograde analysis. *IEEE Computer*, 36(10):26–33, 2003.
29. Schaefer, E. et al. Diablo I. Blizzard Entertainment, 1997. [Online] Available: <http://www.blizzard.com/us/diablo/>, 2009.
30. Tim Schafer, Peter Tsacle, Eric Ingerson, Bret Mogilefsky, and Peter Chan. The Grim Fandango puzzle document. Lucas Arts Confidential, Apr 1996. Public release: Nov 2008.
31. Gurmeet Singh, Carl Kesselman, and Ewa Deelman. Optimizing grid-based workflow execution. *J. Grid Comput.*, 3(3-4):201–219, 2005.
32. Corina Stratan, Alexandru Iosup, and Dick H. J. Epema. A performance study of grid workflow engines. In *GRID*, pages 25–32. IEEE, 2008.
33. Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Conc.&Comp.: Pract.&Exp.*, 17:323–356, 2005.
34. The Entertainment Software Association. 2008 annual report. Technical Report. [Online] Available: <http://www.theesa.com>, Nov 2008.
35. Hong Linh Truong, Schahram Dustdar, and Thomas Fahringer. Performance metrics and ontologies for grid workflows. *Future Gen. Comp. Syst.*, 23(6):760–772, 2007.
36. Gregor von Laszewski and Mihael Hategan. Workflow concepts of the Java CoG Kit. *J. Grid Comput.*, 3(3-4):239–258, 2005.
37. B. S. Woodcock. An analysis of mmog subscription growth. Online Report. [Online] Available: <http://www.mmogchart.com>, 2009.
38. Hiroshi Yamamoto, Nob Yoshigahara, Goro Tanaka, Mine Uematsu, and Harry Nelson. Lunar Lockout: a space adventure puzzle. ThinkFun, 1989.