Universidad Zaragoza
1542

Trabajo Fin de Grado

Diseño e Implementación de un Procesador Hardware Específico para el Juego Blokus Duo

Autor

Alberto Delmás Lascorz

Director

Javier Resano Ezcaray

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

Escuela de Ingeniería y Arquitectura
2014

Diseño e Implementación de un Procesador Hardware Específico para el Juego Blokus Duo

RESUMEN

En el contexto de la competición de diseño del congreso ICFPT 2013, la Universidad de Zaragoza presentó un diseño de un procesador específico de inteligencia artificial para el juego de mesa Blokus Duo. En Diciembre de 2014 la competición vuelve a celebrarse y el objetivo de este proyecto es mejorar el diseño de dicho procesador, que se implementa sobre una FPGA (circuito integrado programable).

Para ello en primera instancia se han investigado las técnicas empleadas por las inteligencias artificiales más efectivas para este tipo de juego (principalmente el software Pentobi), considerando cuáles podrían ser aprovechadas.

También se ha mejorado el diseño hardware inicial aprovechando al máximo el paralelismo a la hora de procesar los tableros. Además se ha actualizado la versión software (que se utilizó el año pasado para diseñar y probar el algoritmo de inteligencia artificial) incluyendo algunas mejoras de las que solo disponía el hardware hasta ahora.

Para agilizar la investigación de mejoras del diseño se crearon herramientas para automatizar tareas repetitivas, permitiendo enfrentar el diseño contra diferentes versiones de si mismo así como contra otros adversarios, y recoger estadísticas de un gran número de partidas sin interacción humana.

Se han desarrollado estrategias para acelerar la búsqueda, implementándolas en la versión software para comprobar su efectividad antes de efectuar la implementación (más costosa) en el hardware. Estas estrategias fueron sopesadas teniendo en cuenta su idoneidad para el diseño hardware, y consiguieron una mejora de velocidad de más de un orden de magnitud.

Finalmente se han investigado otras formas de mejorar la inteligencia sin aumentar el espacio de búsqueda, adaptando ideas de otras implementaciones a los requisitos específicos del diseño hardware (como por ejemplo las restricciones de memoria y cantidad de tiempo fija). Estas mejoras incrementaron la efectividad de la inteligencia artificial significativamente.

El resultado de todo este trabajo es un diseño que funciona entre 10 y 100 veces más rápido que la versión inicial y con algoritmos de inteligencia artificial más potentes que le permiten evaluar mejor las situaciones que se dan durante la partida. Este diseño es competitivo cuando juega contra las mejores aplicaciones software para este juego incluso aunque estas se ejecuten en plataformas mucho más rápidas, con muchos más recursos de memoria, y con un consumo energético varias veces superior.

Como resultado adicional cabe destacar que un artículo redactado sobre las mejoras hardware del diseño realizado fue seleccionado, tras un proceso de revisión por pares, por el congreso ICFPT para la publicación en sus actas.

Índice

Índice de figuras

Índice de tablas

1 Introducción

1.1 Competición de diseño ICFPT 2014

El Congreso Internacional sobre Tecnología Programable (ICFPT) organiza anualmente un concurso de diseño con la temática de utilizar procesadores hardware específicos para resolver determinados problemas, que se implementan sobre un tipo de hardware programable denominado FPGA (Field Programmable Gate Array).

La edición 2013 proponía la creación de un procesador que ejerciese de inteligencia artificial para jugar al juego de mesa Blokus en su variante Duo, con un límite de procesamiento de un segundo por turno. Los concursantes procederían entonces a jugar entre sí en una sesión del congreso para determinar el más efectivo. La universidad de Zaragoza presentó un diseño que obtuvo el cuarto puesto de entre un total de veintiún concursantes.[I]

La edición 2014 se celebrará a mediados de diciembre y continúa con la misma temática, de forma que el objetivo es mejorar el diseño para participar de nuevo en este concurso.[II]

1.2 Blokus Duo

Blokus es un juego de tablero donde cada jugador dispone de 21 piezas (todas las combinaciones posibles de tamaños 1, 2, 3, 4 y 5 con conectividad directa – véase Fig. 1), y debe intentar colocar la máxima cantidad posible sobre el tablero, de acuerdo con las siguientes normas:

- No se puede colocar sobre espacios ya ocupados.
- La pieza debe tocar en diagonal a al menos una pieza del mismo jugador.
- La pieza no puede tocar directamente (por sus lados) a una pieza del mismo jugador.

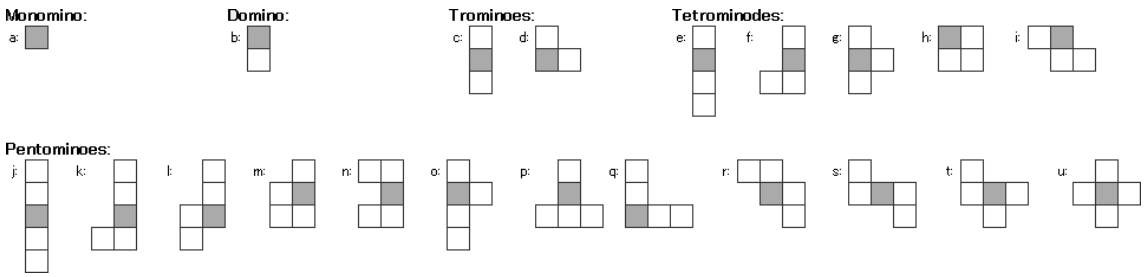


Fig. 1: Piezas disponibles en Blokus

Las piezas pueden rotarse libremente y colocarse al revés. Dependiendo de sus simetrías, algunas piezas pueden colocarse de ocho formas diferentes (véase Fig. 2).

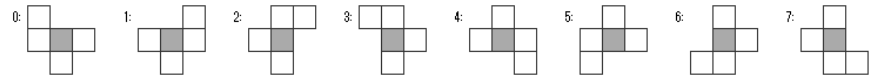


Fig. 2: Rotaciones posibles para la pieza "t"

La variante Duo se juega con dos jugadores en un tablero de 14 × 14 casillas (Fig. 3).

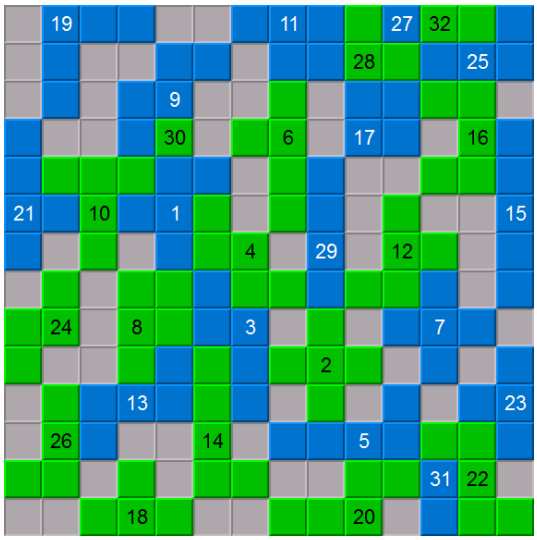


Fig. 3: Ejemplo de partida con los movimientos numerados sucesivamente. La partida ha terminado al no poder ningún jugador colocar más pieza. Ambos jugadores han colocado 16 de sus 21 piezas, pero el jugador azul ha ganado al haber rellenado una casilla más.

1.3 FPGAs

Una FPGA (*Field Programmable Gate Array*) es un circuito integrado compuesto de lógica e interconexiones programables, que permite implementar diseños descritos con lenguajes como VHDL (el utilizado en este proyecto) o Verilog.[III]

Las FPGAs están típicamente compuestas por:

- Bloques lógicos que implementan funciones (normalmente mediante tablas de consulta programables)
- Bloques de memoria interna
- Celdas se entrada/salida para comunicación con el exterior
- Recursos de interconexión, que conectan las salidas de ciertos bloques con las entradas de otros, de acuerdo a la programación

La programación de una FPGA con un diseño preparado es generalmente rápida (del orden de segundos), aunque la síntesis del diseño puede ser muy costosa (del orden de minutos u horas).

1.3.1 Protocolo de comunicación

La FPGA (o ambas FPGAs, en la competición) se comunica con un ordenador personal mediante puerto serie (RS-232C o adaptadores compatibles) que coordina la partida utilizando un protocolo simple donde esencialmente se transmiten las jugadas codificadas en cuatro caracteres ASCII, correspondiendo a ambas coordenadas, el identificador de pieza, y su rotación. El diseño hardware debe mantener internamente el estado de la partida.[IV]

1.3.2 Consumo de energía

Un aspecto interesante de los procesadores programables es su bajo consumo, gracias al cual están generando interés en sistemas embebidos.

La versión hardware no solo es mucho más rápida en términos absolutos (más de un millón de tableros procesaos por segundo, contra alrededor de doscientos mil en el caso del software), si no que también es mucho más eficiente en términos de consumo energético, ya que la FPGA siempre se mantiene por debajo de 10W, mientras que un procesador de propósito general moderno puede llegara consumir cerca de 100W (véase Fig. 4).

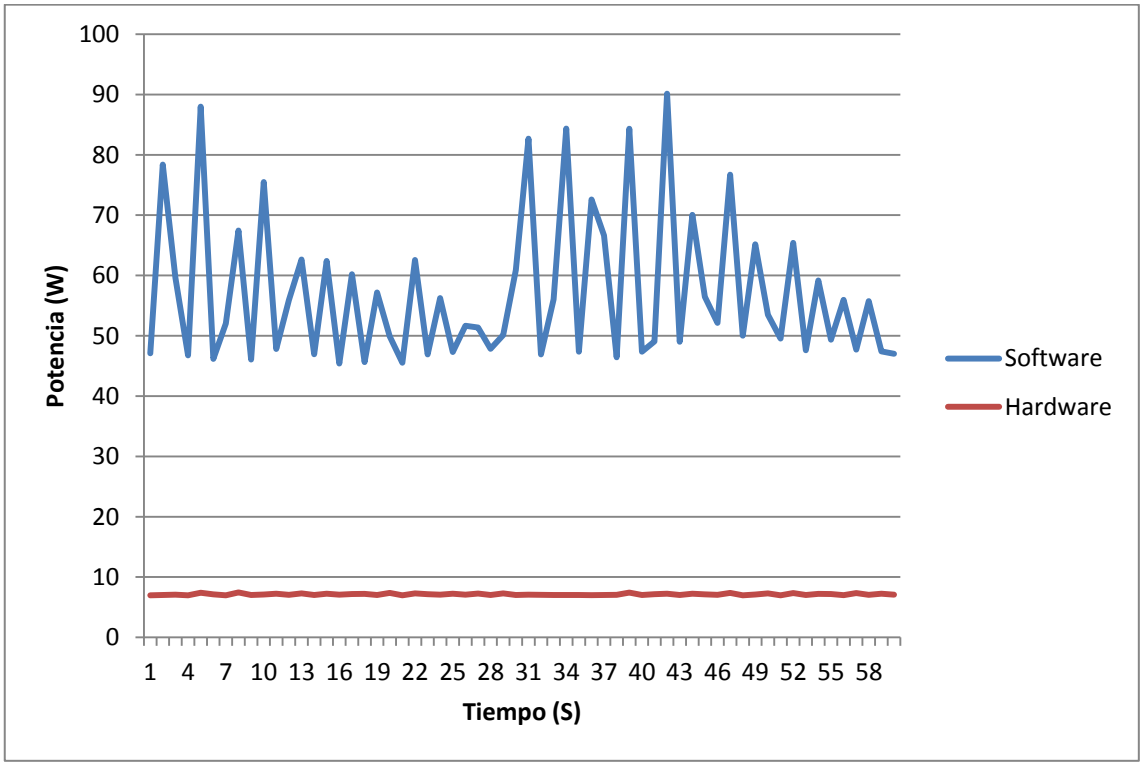


Fig. 4: Consumo de potencia eléctrica durante el transcurso de una partida entre un ordenador personal de sobremesa y una FPGA. La potencia estática de la FPGA es de unos 7W y la dinámica es despreciable, mientras que el ordenador requiere 45W constantemente y el doble al realizar cálculos de forma activa.

2 Estado inicial

El algoritmo de búsqueda empleado en el diseño es Minimax, donde se exploran los diferentes posibles movimientos del jugador y su rival, valorándolos teniendo en cuenta que el jugador debe escoger el movimiento que más le convenga (el de mayor puntuación – *Max*), y su oponente puede escoger, en el peor de los casos, el que menos convenga al jugador (el de menor puntuación – *Min*).[V] Esta búsqueda se acelera mediante una poda alfa-beta.[VI]

Al no conocer a priori el tiempo necesario para una exploración con una profundidad dada, no queda otro remedio que utilizar una búsqueda en profundidad iterativa[VII], es decir, comenzar una búsqueda a baja profundidad (en particular, tanto el software como el hardware empiezan explorando dos niveles, con un coste que rara vez excede una centésima de segundo) e ir aumentando progresivamente la profundidad hasta agotar el tiempo asignado.

La función de coste utilizada valora tanto las piezas colocadas (la puntuación del juego propiamente dicho) como las casillas accesibles por cada jugador (a las que podría llegar con alguna de sus fichas – véase Fig. 5). La accesibilidad da una idea de la movilidad de cada jugador, y penaliza aquellas situaciones que cortan caminos.

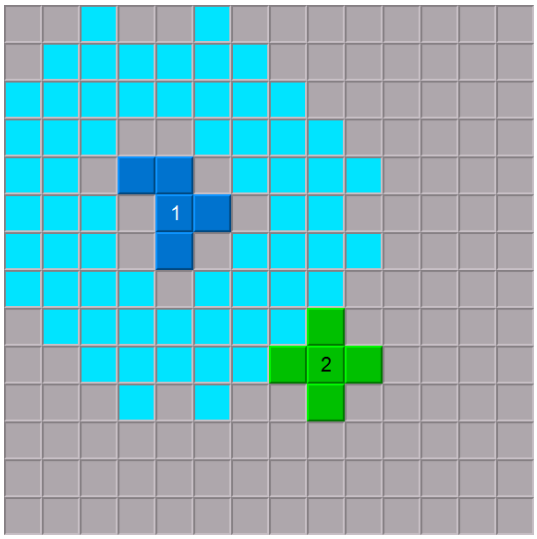


Fig. 5: Ejemplo de accesibilidad calculada para el jugador azul. En la función de evaluación, las casillas tomadas tienen el doble de peso que las accesibles.

Esta función es cara en cómputo pero de alta calidad – tomando únicamente las puntuaciones de las piezas, solo se verían empates durante el comienzo de la partida, ya que ambos jugadores siempre pueden colocar piezas del tamaño máximo durante los primeros turnos.

3 Mejoras de procesamiento

En comparación con la versión del año pasado, la mayor parte de mejoras de procesamiento hardware se centraron en aumentar el paralelismo de la búsqueda, para evaluar un vértice por ciclo (antes hacía falta un ciclo para cada posición de cada una de las rotaciones de cada una de las piezas en cada uno de los vértices, y además la frecuencia de reloj máxima alcanzable era inferior).

El diseño en general y estas mejoras en particular están descritas en el Anexo 1. A parte de replicar unidades funcionales para evaluar varios vértices a la vez, no es fácil extraer más rendimiento del hardware. Por lo tanto, el resto del trabajo se centra en intentar obtener mejores resultados sin aumentar la velocidad de exploración (el número de nodos visitados, o tableros evaluados, por segundo).

Sin la ayuda de la poda alfa-beta, el algoritmo Minimax requiere expandir el árbol de juego completo. Como veremos más adelante, hay formas de realizar la búsqueda de manera más eficiente reordenando los nodos a visitar, pudiendo obviar las visitas a la mayor parte de los nodos, siendo posible determinar por adelantado que no modificarán el resultado de la búsqueda. Para soportar esta reordenación se requieren nuevos módulos hardware, como la memoria de ordenación descrita en la sección *G. Node Reordering* del Anexo 1.

4 Referencias y rivales

4.1 Pentobi

Pentobi[VIII] es una implementación de código abierto de diferentes variantes de Blokus, incluyendo Duo. Su inteligencia artificial presenta un especial interés ya que es extraordinariamente fuerte y razonablemente rápida, así que una de las primeras acciones fue averiguar el algoritmo de búsqueda utilizado.

Se trata de un algoritmo probabilista denominado búsqueda en árbol de Monte Carlo[IX]. Sin entrar en detalles, procede de la siguiente manera:

1. Crear un árbol vacío.
2. Añadir a su raíz todos los movimientos posibles (opcionalmente valorándolos a priori utilizando una función de conocimiento previo[X]).
3. Recorrer el árbol desde la raíz eligiendo nodos de forma aleatoria, pero con un sesgo proporcional a su valoración.
4. Al llegar a una hoja, incrementar su contador de visitas. Si el contador de visitas alcanza un determinado umbral, expandir el nodo (como en el paso 2). En caso contrario, lanzar un *playout* desde ella – una partida completa con movimientos escogidos de forma pseudoaleatoria. Utilizar el resultado del playout para modificar su valoración.
5. Volver al paso 3, hasta que haya transcurrido una cantidad de tiempo determinada.

El sesgo hacia los movimientos más prometedores hace que estos se exploren más asiduamente, invirtiendo en ellos más tiempo de búsqueda. La aleatoriedad permite seguir considerando los movimientos menos prometedores. Además, no hace falta una función de evaluación especulativa explícita, ya que se valoran partidas jugadas hasta el final (aunque sí que es conveniente disponer de una buena función de conocimiento previo para optimizar el proceso).

La búsqueda en árbol de Monte Carlo presenta bastantes dificultades de cara a una implementación en hardware. Es un algoritmo no determinista que depende de la generación de números aleatorios, y requiere una gran cantidad de memoria (en el caso de Pentobi, 768 megabytes – por encima de lo permitido en el concurso, y de lo que tienen muchas FPGAs) a la que accede constantemente en patrones poco predecibles.

Al menos un oponente de la competición de diseño ICFPT 2013 intentó un diseño hardware basado en este algoritmo de búsqueda, aunque no alcanzó la última etapa de la competición.[XI]

4.2 FPGA Blokus Duo Solver

El diseño ganador de la competición ICFPT 2013. A juzgar por los datos públicos, este diseño utiliza una estrategia de búsqueda Minimax sin particularidades, pero con una velocidad de exploración muy elevada.[XII]

5 Herramientas y automatización

Una de las principales dificultades de desarrollo era la inexistencia de automatización en procesos como jugar partidas, tanto en hardware como en software.

En software estaba guiado completamente por código: para analizar un movimiento determinado, había que modificar su código fuente y recompilarlo. Para jugar una partida, había que recompilar tras cada movimiento, así como para cambiar cualquier parámetro de búsqueda.

En el caso de hardware, se podía utilizar la interfaz por puerto serie, pero esto requería codificar manualmente (utilizando una tabla) los movimientos e introducir su correspondiente código, uno por uno. Esta tarea repetitiva podía consumir varios minutos por partida.

Para solucionar estos problemas, se modificó el software para leer los parámetros de entrada desde fichero, y se adaptó el programa de comunicación por puerto serie para permitir jugar partidas automáticamente con cualquier combinación posible:

Software	vs	Software
Hardware		Hardware
Pentobi		Pentobi

Esto también requiere conversiones de formato, pues la competición trata fichas enteras y Pentobi trabaja con casillas individuales, además de utilizar otro sistema de coordenadas.

Las partidas automáticas se almacenan en disco para permitir su análisis posterior. Es posible hacer jugar a la FPGA cientos partidas de forma automática para probar cambios de parámetros, y después ver un sumario de resultados y reproducir partidas individuales.

Otra ventaja es pode utilizar el análisis de partidas de Pentobi, que valora cada movimiento, para poder describir puntos débiles. Véase Fig. 6 para un ejemplo.

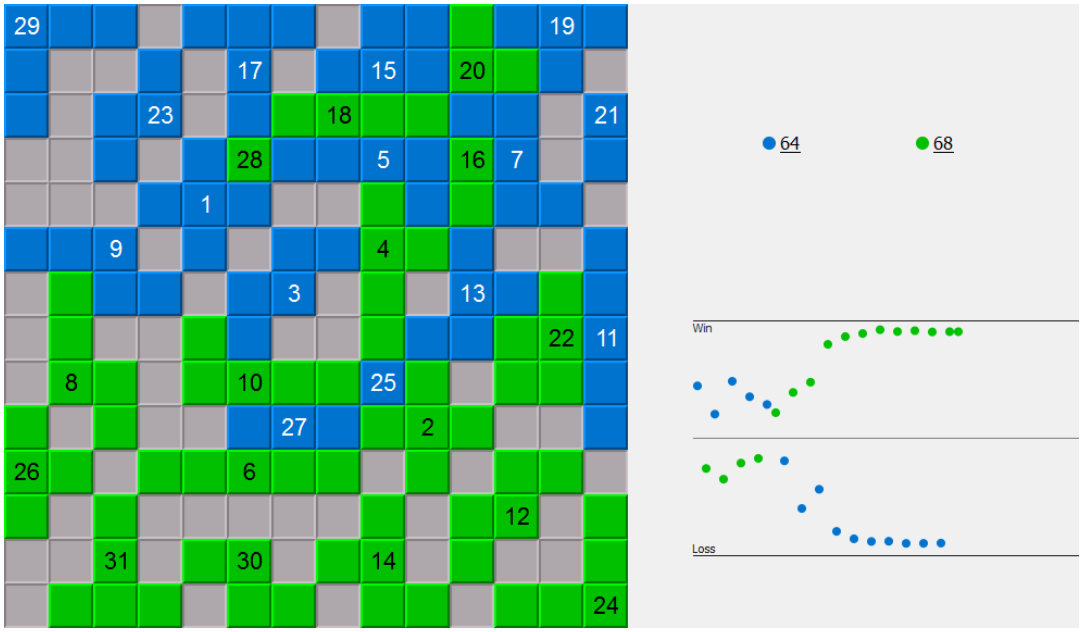


Fig. 6: En esta partida, el movimiento 11 ha sido especialmente problemático para el jugador azul.

5.1 Depuración y prevención de errores

Una ventaja fundamental de mantener una versión software que implementa exactamente las mismas técnicas que la versión hardware es que resulta tremendamente útil para depurar. En ese sentido se realizó un esfuerzo importante para que ambas versiones explorasen las mismas situaciones en el mismo orden. Y como se utilizan algoritmos deterministas, ambas debían generar las mismas salidas. Depurar un diseño hardware es extremadamente complejo, especialmente cuando se procesan millones de tableros en cada movimiento. El desarrollo de la versión software ha sido clave para encontrar errores mucho más rápido.

Se han realizado otros esfuerzos para prevenir errores en el diseño hardware como desarrollar pequeños programas que generasen automáticamente secciones de código especialmente tediosas, o que comprobasen si los datos almacenados en las memorias del diseño hardware eran correctos. Estas herramientas permitieron encontrar algunos errores que se manifestaban con poca asiduidad por lo que no habían sido detectados previamente.

6 Mejoras de poda hardware implementadas en software

La versión inicial del software carecía de algunas podas importantes que se implementaron tarde en la versión hardware. Para acelerar la experimentación y obtener resultados equivalentes a los del hardware, era necesario incorporar las mejoras ya presentes en el mismo a la versión de software.

6.1 Solapamiento

Una forma de descartar movimientos poco útiles al comienzo de las partidas es requerir su colocación en el área accesible por ambos jugadores, que es la que interesa ocupar siempre que sea posible (jugar en dirección al oponente). Para ello se calculan las casillas accesibles por el oponente y se descartan los movimientos que no ocupen un número suficiente de dichas casillas. El umbral disminuye progresivamente durante la partida.

Por ejemplo, en la situación ilustrada en la Fig. 7, el jugador azul tiene 452 movimientos posibles. Añadiendo el requisito de ocupar, al menos, 3 casillas accesibles al jugador verde, el número de movimientos a considerar se reduce a 154.

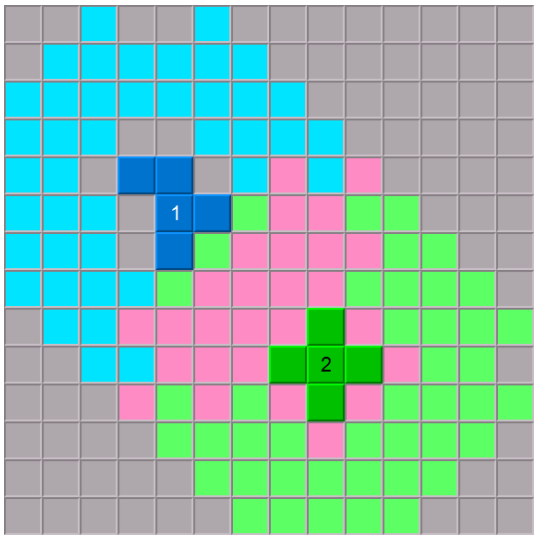


Fig. 7: Accesibilidades de ambos jugadores y zona solapada en rosa. La zona solapada también considera como accesibles las casillas en contacto directo con el adversario, aunque éste no puede colocar piezas ahí.

Esta técnica está descrita en la sección *B. Move discard* del Anexo 1.

6.2 Ordenación de vértices

Por razones que se expondrán más adelante, conviene realizar la exploración comenzando por los mejores movimientos. En vez de realizar la exploración de arriba abajo y de izquierda a derecha, el hardware comienza en el centro y se expande en forma de rombo hacia los bordes.

En la mayoría de los casos esta reordenación no acelera la búsqueda más de un 5%, pero su implementación no tiene coste en hardware y en algunos casos extremos puede llevar a búsquedas un 40% más rápidas.

6.3 Otras optimizaciones de software

Al analizar el software utilizando un perfilador se detectaron varios puntos calientes donde pequeños cambios (eliminación de saltos condicionales poco predecibles, o reordenación de condicionales) resultaron en mejoras de velocidad del 30%. Aunque es irrelevante para el diseño hardware, esta aceleración ayuda a agilizar la experimentación con el software lo cual es muy importante porque los análisis realizados para evaluar distintas opciones requerían varias horas.

Gracias a la inclusión de todas las técnicas de optimización que sólo estaban inicialmente en la versión hardware y a las optimizaciones descritas previamente se consiguió acelerar la experimentación con el software aproximadamente dos órdenes de magnitud, permitiendo exploraciones con profundidades similares a las alcanzables por la implementación hardware.

7 Mejoras de poda prototipadas y evaluadas en software

7.1 Problemas de la exploración iterativa

La exploración iterativa utilizada en el diseño tiene dos problemas principales:

1. El desaprovechamiento de los primeros niveles. Como el número de nodos a visitar crece exponencialmente con el nivel, la cantidad de tiempo perdida en los primeros niveles es despreciable. Por ejemplo, en la situación ilustrada en la Fig. 8, el jugador verde tiene 128 movimientos a considerar, aun considerando solo las piezas de tamaño 5 y la poda por accesibilidad descrita anteriormente. Sin otras mejoras, cada nivel costaría 128 veces más tiempo que el anterior, y por lo tanto la pérdida total sería de $\sum_{n=1}^{l-1} \frac{1}{128^n}$, donde l es el último nivel explorado. La suma de esta serie geométrica es de menos de un 1%.

   Como veremos, gracias a las optimizaciones de búsqueda, el factor de ramificación en la práctica es bastante más pequeño, pero rara vez es inferior a 10 (que supondría una pérdida total de menos del 12%).

2. El desaprovechamiento del último nivel. Por el mismo razonamiento, si la búsqueda se detiene tras una cantidad de tiempo fija, la mayor parte de este habrá sido empleado en un nivel que no se ha conseguido terminar. Normalmente dicho nivel se descarta y se procede a utilizar como resultado el obtenido en el anterior.

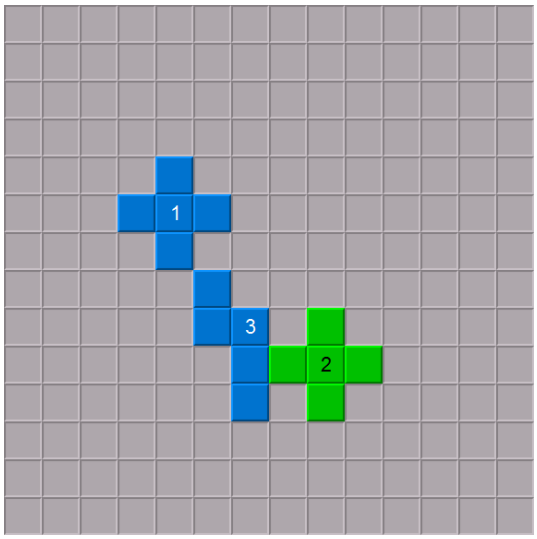


Fig. 8: Situación temprana en una partida

Pero también tiene ventajas: una de las más grandes es poder utilizar los datos obtenidos en los niveles superiores para acelerar la búsqueda en los más profundos, y a la vez solucionar parcialmente el segundo problema.

7.2 Ordenación de los nodos

7.2.1 Motivación: Poda alfa-beta

En general, la complejidad (cantidad de nodos visitados) de una búsqueda en profundidad es $O(b^d)$ donde d es la profundidad y b es el factor de ramificación. La poda alfa-beta reduce esta complejidad, pero su efectividad depende del orden de exploración (véase Fig. 9).

En el mejor de los casos, cuando la exploración se realiza de mejor a peor sucesor, el espacio de estados explorado se ve reducido a $O(b^{d/2})$ o, equivalentemente, se pueden explorar el doble de niveles empleando la misma cantidad de tiempo.

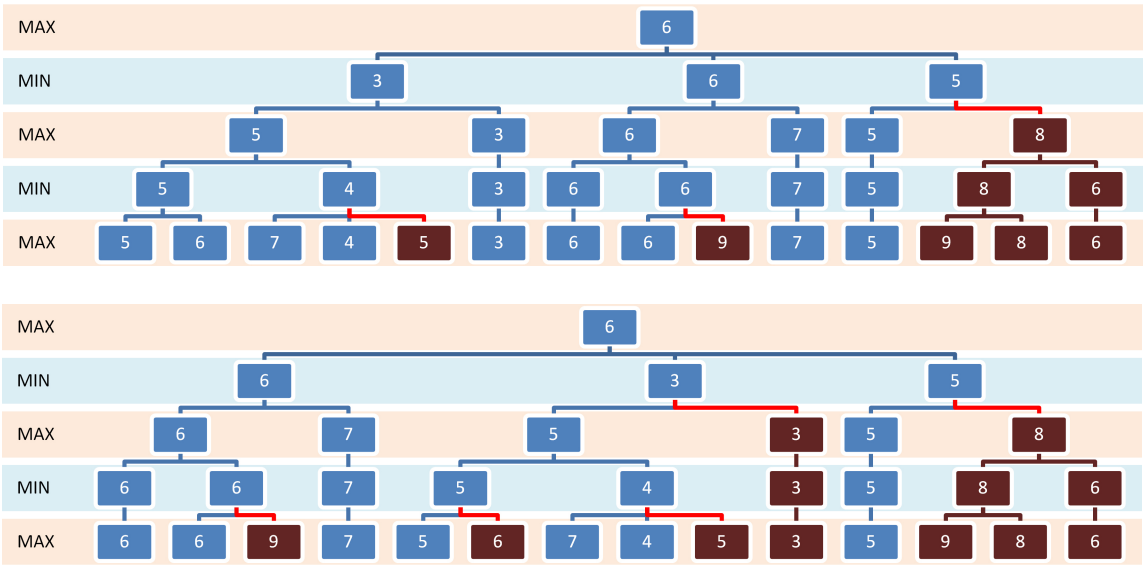


Fig. 9: Ejemplo de poda alfa-beta con diferentes ordenaciones. En el segundo caso se podan un 50% más de nodos. El efecto es más extremo con mayores factores de ramificación.

7.2.2 Motivación: Aprovechamiento del último nivel

Al interrumpir (por límite de tiempo) una exploración con profundidad *n*, si se ha explorado el mejor nodo resultado de la exploración a profundidad *n–1*, es posible devolver el mejor nodo conocido hasta el momento en vez de descartar el nivel completo, ya que si algún otro nodo ha obtenido mejor puntuación significa con certeza que es más valioso.

Generalizando esta idea, explorar los nodos de mejor a peor según el nivel previo maximiza las posibilidades de encontrar el mejor en el nivel actual, en caso de que la búsqueda se interrumpida en un momento arbitrario.

Mov.	Nivel 2	Nivel 3	Nivel 4	Nivel 5
43l0	22	32	18	35
32l4	21	31	17	33
c6k5	20	43	16	36
43p2	15	26	17	36
33m4	15	26	17	35
33l7	15	25	9	35
c6n0	15	42	12	33
c6p4	14	42	12	34
33m7	14	24	11	34
34l3	14	25	15	34

Tabla 1: Movimientos y valores

En la Tabla 1 pueden verse los valores de unos movimientos determinados a diferentes profundidades (se muestran los 10 primeros de 216). Como puede observarse, el considerado mejor a nivel *n* (marcado en amarillo) está siempre entre los 5 primeros del nivel *n–1* (marcados en naranja).

Aunque el movimiento mejor no siempre se encuentra tan arriba, normalmente hay una correlación muy fuerte entre niveles sucesivos. En cualquier caso el resultado del aprovechamiento de un nivel parcial nunca es peor que descartar dicho nivel por completo.

7.2.3 Motivación: Descarte directo de movimientos pobres

Como se ha visto en la sección anterior, una primera exploración puede dar información sobre la calidad de los movimientos a priori – permitiendo descartar completamente los de menor puntuación. Aunque esto reduce notablemente el factor de ramificación, la poda alfa-beta hace que la ganancia de velocidad real sea moderada (los mejores movimientos son los más costosos de explorar, y los peores son podados rápidamente). Además, descartar movimientos a ciegas potencialmente puede alterar los resultados, al contrario que las otras técnicas.

Una variante es ir descartando progresivamente más movimientos conforme la profundidad aumenta.

7.2.4 Ordenación completa: dificultades en hardware

La solución ideal sería almacenar el árbol completo de búsqueda en memoria y mantenerlo ordenado para poder profundizar la búsqueda de forma óptima. Sin embargo, la cantidad de memoria requerida crece (como el número de nodos explorado) de forma exponencial, siendo intratable por las mismas razones que la búsqueda en árbol de Monte Carlo.

Además, es conveniente almacenar los datos en las memorias internas de la FPGA: esto permite acceder a ellos de forma inmediata, y mantenerlos siempre ordenados (ver Anexo 1 sección *G. Node Reordering*), sin pagar la latencia de acceso a una memoria externa. La FPGA dispone de una memoria interna del orden de varios millones de bits: más que suficiente para almacenar el primer nivel, pero insuficiente para almacenar tres o más niveles salvo en casos muy ventajosos.

Por estas razones es necesario implementar una versión parcial en hardware, así que se decidió prototipar en software diferentes implementaciones de complejidad ascendente para ver qué mejoras ofrecían y si valía la pena su coste en hardware (tanto en recursos de FPGA como en tiempo de implementación).

7.2.5 Ordenaciones parciales

Se mejoró el software para poder elegir la cantidad de niveles almacenados con orden, además de la posibilidad de almacenar la mejor cadena de movimientos (el movimiento de cada nivel que lleva a la puntuación máxima en la raíz) para cada movimiento de primer nivel.

Los requisitos de memoria aumentan de forma exponencial con el número de niveles, pero en el caso de la mejor cadena el aumento es lineal, siendo perfectamente razonable almacenar todas las cadenas (con una ramificación de 512 y una profundidad máxima de 8, es necesario almacenar 4096 movimientos de 16 bits cada uno).

En la Tabla 2 al final de la sección pueden verse las mejoras obtenidas gracias a estas modificaciones en el orden de exploración.

7.2.6 Tabla *hash* de podas

Ante la impracticabilidad de almacenar todo el árbol de exploración en memoria, una idea para aprovechar los recursos limitados de los que dispone la FPGA es almacenar los movimientos que han sido útiles para realizar podas – los que producen una poda inmediatamente después de su exploración.

Para almacenarlos de forma que la información sea útil, se debe utilizar como clave el estado actual de la exploración (la cadena de movimientos desde la raíz hasta el de nivel inmediatamente inferior), y como dato (o valor) debe almacenarse el movimiento en cuestión (véase Fig. 10).

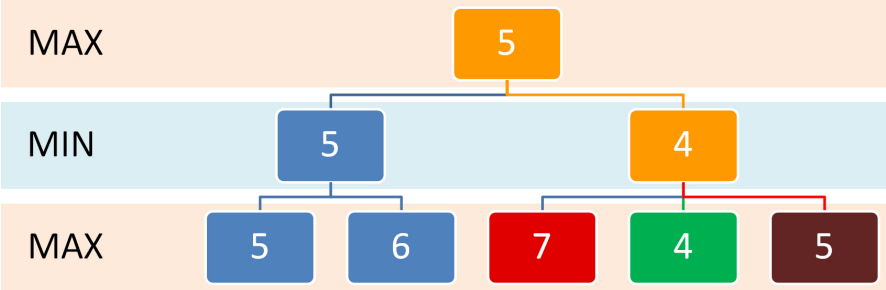


Fig. 10: Esquema de información a almacenar. La clave es el camino marcado en color anaranjado, y el dato es el nodo (verde) que ha servido para realizar la poda. Cuando se vuelva a visitar este estado, se comenzará por el nodo verde, de forma que tanto el rojo como el marrón serán podados (dos en lugar de uno).

Además, es deseable poder acceder y actualizar movimientos en tiempo constante (el hardware puede expandir un nodo por ciclo), y no es necesario un almacenamiento sin pérdidas, ya los datos se utilizan únicamente para acelerar la búsqueda.

Por lo tanto, una estructura adecuada para el almacenamiento sería una tabla *hash* oportunista, con direcciones dadas por la clave y almacenando únicamente el valor, sin control de colisiones. Es decir: una estructura donde, tomando la clave como entrada, se determina una posición para almacenar el dato a partir de una función (generalmente no inyectiva) que intente repartir uniformemente las colocaciones, sin intentar solucionar el caso en el que dos claves acaban correspondiendo a la misma posición.

Las ventajas de esta estructura son múltiples: todo el espacio de memoria puede ser utilizado para almacenar movimientos (y, en este caso, las claves ocuparían más espacio que los movimientos en cuestión), y el rendimiento de búsqueda y actualización es excelente.

Las desventajas son las pérdidas por colisiones y la posibilidad de leer movimientos que no corresponden, ya sea por colisión o por edad (es caro limpiar la memoria – los bits de validez – entre exploraciones, aunque es posible almacenar un número generacional que se puede incrementar para borrar implícitamente la tabla. Con 2 bits por entrada pueden evitarse colisiones entre 4 exploraciones sucesivas).

Ambas pérdidas son asumibles. Si quisiésemos evitarlas almacenando las claves y utilizando encadenado, se perdería la mayor parte de la memoria almacenando las claves (por ejemplo, en una poda en profundidad 4, la clave necesitaría 48 bits, y el dato 16), llevando a la necesidad de emplear una tabla más pequeña que no compensaría el aumento de precisión en la búsqueda. Además, el rendimiento de las dos operaciones básicas decaería.

Sobre la posibilidad de leer movimientos no correspondientes, en la práctica no es un problema ya que los movimientos almacenados son valiosos por definición. Si son válidos (cosa que hay que comprobar) seguirá siendo ventajoso explorarlos primero. En el peor de los casos solo habrá un cambio en el orden de exploración.

El hecho de comprobar si son válidos sí que supone una complejidad extra en la implementación: hasta ahora teníamos la certeza, por construcción, de que todos los movimientos considerados cumplían ciertas condiciones (que su pieza todavía no había sido utilizada, y que hacían contacto esquina con esquina). Leyendo movimientos de la tabla *hash* estas garantías ya no existen y hay que comprobarlas explícitamente.

Llegado este punto, es necesario diseñar la función de hash propiamente dicha y la política de remplazo en caso de colisiones.

Para la función de hash (y el almacenamiento de los datos), podemos representar un movimiento en 16 bits como:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
coordenada X				**coordenada Y**				**pieza**					**rotación**		

Las probabilidades de algunos de estos bits no son uniformes, pero esto no es muy importante gracias a las siguientes transformaciones.

Para reducir una cadena de movimientos de longitud variable a un hash de tamaño fijo uniformemente distribuido se puede emplear la estrategia de combinar cada uno de los movimientos al hash utilizando alguna transformación en cada paso. Con esta idea, dos funciones de deferente complejidad fueron diseñadas.

Baja complejidad:

```
hash <= 0
para cada [nivel]:
    hash <= hash ROL 3
    hash <= hash XOR movimiento[nivel]
devolver hash
```

Media complejidad:

```
hash <= 0
para cada [nivel]:
    hash <= hash * 31
    hash <= hash XOR movimiento[nivel]
devolver hash
```

Donde **hash** tiene tantos bits como sea necesario para direccionar la tabla (16 por ejemplo para una tabla de 128K con 65.536 entradas). **ROL** es una rotación a izquierda, y **XOR** denota un Ó exclusivo.

En el primer caso, la rotación de 3 bits intenta que los bits del resultado que contenían menos información (por ejemplo, los más significativos de las coordenadas) se superpongan con los que contienen más información. Este valor es el que mejor funciona en la práctica, ya que permite una buena distribución con cadenas de tan solo tres movimientos.

La segunda función está inspirada en la función `hashCode()` de la clase `java.lang.String`, que opera sobre cadenas de texto representadas por caracteres de 16 bits.[XIII]

Para evaluar estas funciones se analizó un caso real almacenando la información obtenida de analizar un movimiento difícil a profundidad 4. En dicho caso se visitan 915.172 nodos y se producen 25.868 podas, que son almacenadas en una tabla de 65.536 entradas. Si la función hash las distribuyera uniformemente, cabría esperar 4.495 colisiones, de acuerdo a la fórmula

$$n - m\left(1 - \left(\frac{m-1}{m}\right)^n\right)$$

donde *n* es el número de datos a almacenar y *m* es el tamaño de la tabla.

Experimentalmente vemos que la primera función genera 5.391 colisiones, un 20% más de lo esperado. La segunda genera 4.530 colisiones, que es prácticamente lo esperado – un resultado excelente.

Ambas son fáciles de implementar en hardware. La multiplicación de la segunda es siempre por la misma constante (0x11111) y a pesar de sus definiciones iterativas es fácil computarlas en un ciclo para los casos relevantes (cadenas de longitud inferior a 10).

Sobre la política de remplazo, intuitivamente cabría pensar que es más beneficioso respetar a los movimientos introducidos primero, que suelen ser los más valiosos al realizar la búsqueda de mejor a peor y de nivel inferior a superior. La otra opción es el remplazo incondicional de los movimientos previos. En software es fácil probar las dos estrategias, pero en hardware la dificultad del borrado de la tabla (aún con contadores generacionales) favorece la segunda.

Para evaluar la política de remplazo, se midió el número de nodos visitados en una búsqueda de profundidad 5 utilizando los datos recopilados a profundidades 3 y 4, consistentes en 26.784 podas. El no remplazo (“el primero se queda”) lleva a visitar 26.983.161 nodos en total. El remplazo incondicional incrementa el número a 26.783.242, confirmando las sospechas. Por suerte, el empeoramiento es inferior al 1%, de forma que la política de remplazo no tiene un gran efecto en la práctica.

7.3 Resultados de las mejoras de poda

Para evaluar las diferentes mejoras, se cuantificó su efecto sobre el número de nodos visitados en una búsqueda difícil de una situación de comienzo de partida (Fig. 11). Hay un gran aumento en la ramificación a profundidad 5 ya que en este punto se desactiva la poda por solapamiento.

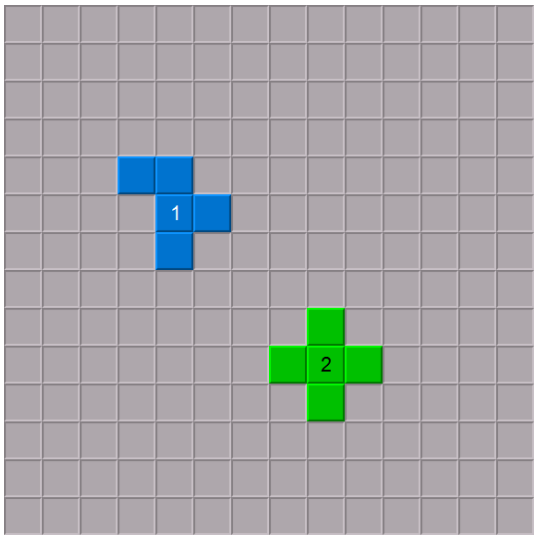


Fig. 11: Situación de comienzo de partida

La estrategia empleada consiste en una exploración sin poda a profundidad 2 (rápida, de 30.073 nodos en todos los casos), seguida por exploraciones podadas con profundidad sucesivamente ascendiente. La única diferencia entre los experimentos es la ordenación de los nodos, que afecta a la efectividad de la poda alfa-beta. El resultado de las búsquedas a una cierta profundidad siempre es el mismo.

Niveles ordenados	Cadena de mejores	Tabla hash de podas	Coste profundidad 3	Coste profundidad 4	Coste profundidad 5
0	No	No	1.010.523	48.832.391	###.###.###
0	No	Sí	1.010.523	19.676.616	###.###.###
1	No	No	216.488	11.345.945	###.###.###
1	Sí	No	80.962	1.334.502	256.517.641
1	No	Sí	216.488	10.583.421	###.###.###
1	Sí	Sí	80.962	1.231.619	181.486.167
2	No	No	80.859	1.613.885	195.510.471
2	Sí	No	80.859	1.247.743	167.478.383
2	No	Sí	80.859	1.503.307	133.731.790
2	Sí	Sí	80.859	1.224.539	118.295.636

Tabla 2: Resultado de las mejoras de poda. Nota: las cifras no indicadas son superiores a 10^9 y se consideran impracticables.

Todas las mejoras ayudan en mayor o menor medida (véase Tabla 2). Las más importantes son las ordenaciones de los primeros niveles y las cadenas de mejores, aunque debido a solo cubren parte del árbol su efectividad decrece conforme aumenta la profundidad de la búsqueda, ya que aceleran una parte proporcionalmente menor. La tabla hash tiene un alcance más generalizado y ayuda a más profundidad, aunque su capacidad limitada también pone límites.

La combinación de todas las mejoras acelera la búsqueda entre uno y dos ordenes de magnitud, a cambio de aumentar la complejidad de la implementación y su uso de memoria (que es el factor limitante en el diseño hardware).

Llegado este punto, los beneficios obtenidos a cambio de un determinado esfuerzo comienzan a decaer rápidamente: mejoras de decenas de puntos porcentuales parecen deseables pero no van a conseguir incrementar la calidad de juego de una manera apreciable. Para aumentar la profundidad de exploración hace falta un factor de reducción en coste de varios enteros.

Además, conforme se reduce la distancia a la ordenación óptima, técnicas que serían formidables por si solas comienzan a dar resultados pobres. Por poner un ejemplo completo, la tabla hash reduce el espacio de búsqueda hasta un 60% aplicándose al vacío, pero junto con las otras mejoras su aportación se ve reducida a un 30%.

7.3.1 Nota sobre la combinación de poda alfa-beta con ordenación

Un detalle a tener en cuenta al realizar una exploración iterativa en profundidad es que los datos obtenidos utilizando el algoritmo de poda alfa-beta son de peor calidad que los obtenidos sin poda. La poda alfa-beta garantiza que se obtendrá el mismo nodo de mayor valor como resultado de la búsqueda, pero en general las puntuaciones obtenidas para el resto de los movimientos pueden ser superiores a sus puntuaciones reales (la búsqueda se detiene en cuanto se determina que su puntuación no puede superar a la mejor conocida hasta el momento, pero antes de calcularla con exactitud).

Además, puede ocurrir que algunos nodos a profundidad 2 o superior ni siquiera se visiten, necesitando un trato especial de cara a la ordenación.

Estas imprecisiones hacen que los datos obtenidos, de cara a la ordenación para las búsquedas sucesivas, sean de menos calidad. Este efecto es cuantificable, como puede verse en la Tabla 3.

Nivel	Poda	Tiempo	Diferencia	Nivel	Poda	Tiempo
2	No	0,132 s	0 s	2	No	0,132 s
3	Sí	0,318 s	57,974 s	3	No	58,292 s
4	Sí	5,081 s	-0,583 s	4	Sí	4,498 s
5	Sí	778,314 s	-10,8 s	5	Sí	767,514 s

Tabla 3: Empeoramiento por poda sucesiva

A pesar de que la poda afecta a los niveles posteriores, la pérdida de tiempo a nivel tres no compensa las ganancias posteriores. Por lo tanto, la estrategia empleada consiste en realizar una exploración sin poda a nivel 2 (para obtener unos datos iniciales de buena calidad en un tiempo despreciable) y continuar con siempre con poda a partir de ahí.

8 Mejoras de la función de evaluación: *playouts*

A pesar de las mejoras obtenidas en la poda, que permiten profundizar más en la búsqueda Minimax en el mismo periodo de tiempo, el diseño todavía presentaba debilidades en la fase inicial-media de la partida, donde raramente puede llegarse a explorar una profundidad de más de cuatro a seis movimientos, en contraste con la búsqueda de Monte Carlo que siempre profundiza hasta el final de la partida.

Una solución para mejorar la eficacia de juego sin aumentar la profundidad de exploración es mejorar la calidad de la función de evaluación. Hay algunas heurísticas posibles típicas de juego de tablero como por ejemplo la distancia al centro de las piezas. Dentro de la función de evaluación actual también hay parámetros modificables como el peso relativo de las piezas (entre sí) y de la accesibilidad, así como variantes donde se tiene en cuenta el número de piezas diferentes con las que se puede llegar a las posiciones accesibles.

Desafortunadamente tras experimentar con estos parámetros no se encontró ninguna mejora concluyente (véase Tabla 4).

	Victorias	Empates	Derrotas
Versión de control	44	10	46
Piezas doble valor	52	4	44
Piezas cuádruple valor	20	0	80
Accesibilidad triple valor	53	3	44
Piezas valor exponencial, accesibilidad alta	23	7	70
Piezas valor exponencial, accesibilidad media	21	6	73
Piezas valor exponencial, accesibilidad baja	14	4	82
Accesibilidad por piezas diferentes	0	0	100

Tabla 4: Partidas contra la versión de control de sí mismo

Nótese que variaciones del orden de un 10% pueden considerarse ruido a efectos prácticos (habitualmente desaparecen o cambian de sentido al repetir el experimento).

La falta de profundidad en la búsqueda es uno de los problemas detectado observando el comportamiento en partidas reales. La imposibilidad de considerar situaciones futuras trasciende en un comportamiento voraz donde se favorecen las ganancias a corto plazo en detrimento estrategias superiores pero que tardan más en dar resultados evaluables por la función de evaluación.

Pensando en como mejorar la prognosis de la función de evaluación y teniendo en cuenta las ideas aplicadas a la búsqueda de Monte Carlo, se prototipó en software una búsqueda basada en *playouts*, como adición o sustituto a la función de evaluación basada en piezas más accesibilidad. El esquema de dicha función es:

```
Mientras que algún jugador pueda poner una pieza:
  Buscar el mejor movimiento del siguiente jugador que pueda colocar
  Realizar ese movimiento
  Evaluar el estado del tablero con un determinado peso
```

Como ejemplo gráfico ilustrado en la Fig. 12, dada la situación a evaluar ilustrada arriba a la izquierda, el *playout* resultante puede verse a su derecha. Tras utilizarlo para evaluar la situación, se repite el procedimiento para el siguiente movimiento a procesar (abajo a la izquierda), que tendrá su correspondiente *playout* (abajo a la derecha).

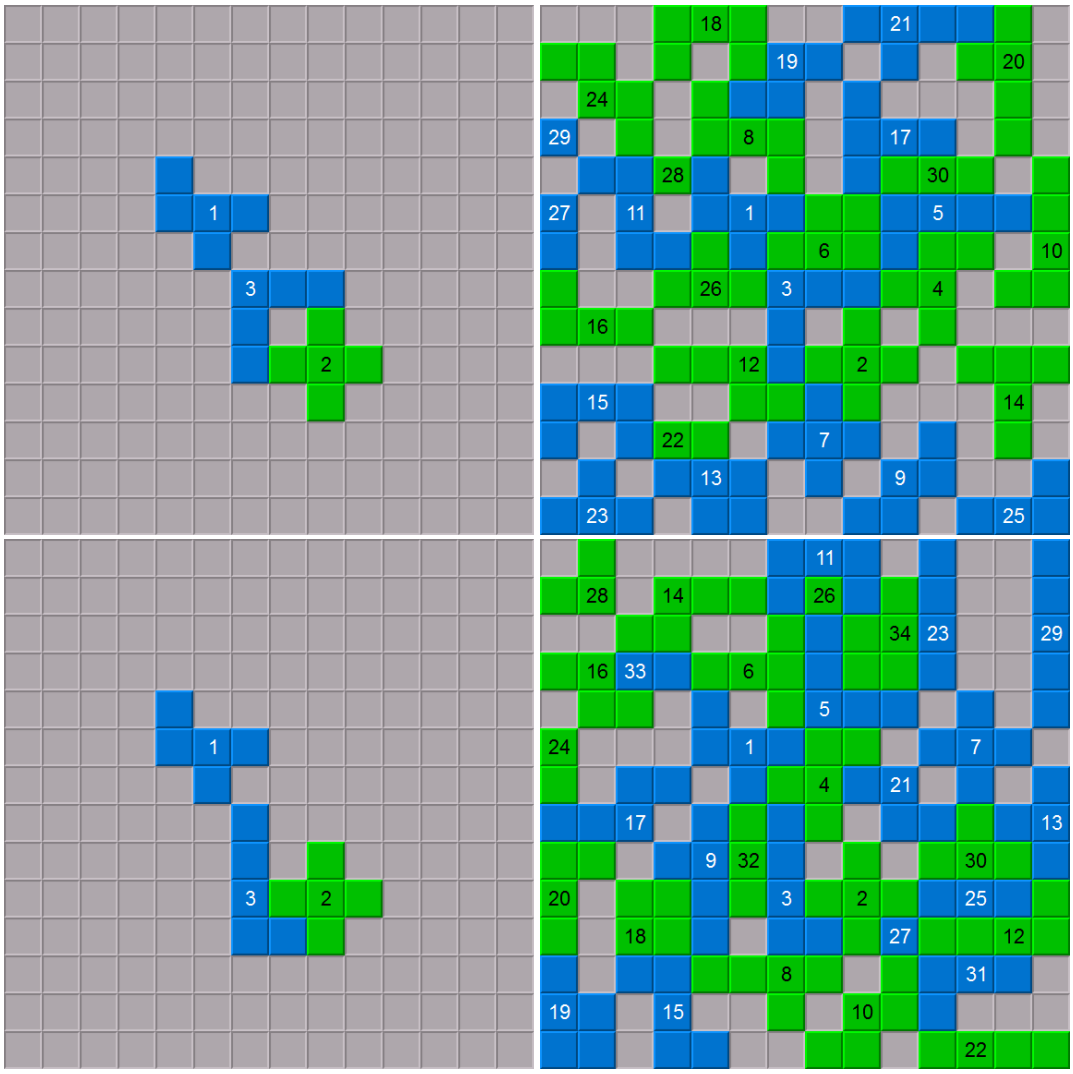
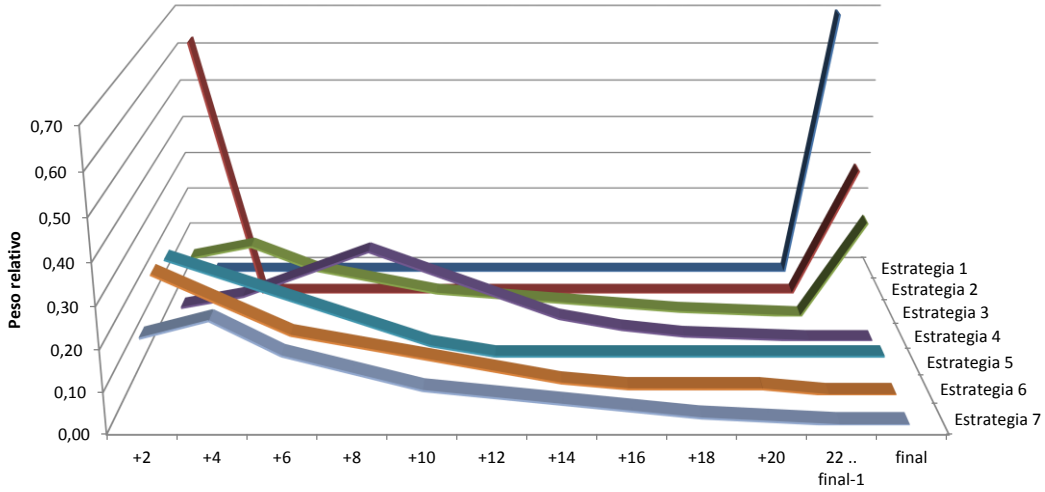


Fig. 12: Dos movimientos con sus respectivos *playouts*

Un parámetro configurable es la profundidad de la búsqueda del siguiente movimiento, que no tiene por qué ser 1 (una búsqueda a profundidad 2 con las mejoras de poda descritas anteriormente es unas diez veces más lenta que la correspondiente búsqueda a profundidad 1). Esto permite ajustar el coste global contra la calidad de la prognosis.

Una vez terminado el proceso y obtenida la valoración, esta partida jugada hasta el final se descarta. Como el diseño es determinista, al contrario que en Monte Carlo, solo se dispone de una única partida (*playout*) por situación del tablero a evaluar.

Hay que decidir que hacer con dicha partida. La solución propuesta consiste en evaluar el tablero obtenido cada dos movimientos jugados en el *playout* y multiplicar esa valoración por un factor que varía en función de la profundidad. El ajuste de estos factores no es evidente por lo que se diseñaron siete estrategias de ajuste distintas que se muestran en la Tabla 5.



	+2	+4	+6	+8	+10	+12	+14	+16	+18	+20	22 .. final-1	final
Estrategia 7	0,21	0,25	0,17	0,12	0,08	0,07	0,05	0,03	0,02	0,01	0,00	0,00
Estrategia 6	0,29	0,22	0,15	0,12	0,09	0,06	0,03	0,01	0,01	0,01	0,00	0,00
Estrategia 5	0,27	0,22	0,16	0,11	0,05	0,03	0,03	0,03	0,03	0,03	0,03	0,03
Estrategia 4	0,09	0,12	0,17	0,23	0,17	0,12	0,06	0,03	0,01	0,01	0,00	0,00
Estrategia 3	0,16	0,19	0,12	0,09	0,06	0,05	0,04	0,02	0,01	0,01	0,00	0,25
Estrategia 2	0,67	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,33
Estrategia 1	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00

Tabla 5: Estrategias de valoración de *playouts*

La valoración se hace cada dos movimientos para evitar cambios bruscos (si un jugador ha colocado una pieza más que el otro, su puntuación va a ser bastante superior hasta que el segundo responda).

Utilizando dichas tablas, se procedió a simular partidas tanto contra Pentobi (nivel 6) como contra una versión del software que no utiliza *playouts*, con una profundidad de exploración equivalente (en software el coste de evaluación se incrementa en un factor inferior a 10 por lo que para poder comparar el rendimiento es necesario detener la exploración antes). Los resultados pueden consultarse en la Tabla 6.

	Contra versión sin *playouts*			Contra Pentobi nivel 6		
	Victorias	Empates	Derrotas	Victorias	Empates	Derrotas
Estrategia 1	28	1	71	14	4	82
Estrategia 2	62	3	35	35	4	61
Estrategia 3	80	2	18	36	4	60
Estrategia 4	59	3	38	47	8	45
Estrategia 5	72	8	20	48	7	45
Estrategia 6	83	4	13	51	6	43
Estrategia 7	78	2	20	52	5	43

Tabla 6: Resultado de las diferentes estrategias de valoración de *playouts*

Analizando estos datos se observan una tendencia clara: es ventajoso dar mayor peso a los movimientos más cercanos (no es de extrañar pues son más certeros que los predichos a gran distancia). En particular, considerar la situación final (como hace Monte Carlo) no da buenos

resultados. Esto tiene además la ventaja añadida de que es posible interrumpir la evaluación del *playout* de forma prematura (tras, por ejemplo, 8 movimientos con sus respuestas), acelerando el proceso de búsqueda (aunque los primeros movimientos son los más caros).

Las mejores estrategias (6 y 7) consiguen ganar consistentemente contra la versión sin *playouts*, y mejoran los resultados contra Pentobi (que eran 39 – 7 – 54 en este caso).

9 Tabla de aperturas

Pentobi, como la mayoría de inteligencias artificiales de juegos de tablero, incluye un árbol de aperturas (Fig. 13), que ayuda a tomar mejores decisiones en los cruciales primeros momentos del juego, donde además el factor de ramificación es enorme y es difícil realizar búsquedas profundas (el diseño hardware no suele llegar a analizar seis movimientos por delante durante la fase de apertura).

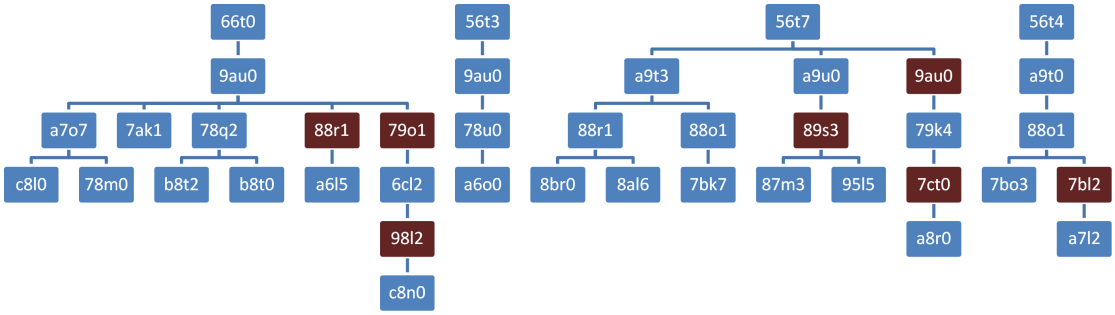


Fig. 13: Árbol de aperturas (parcial). Los nodos marcados en rojo son no deseables.

Es fácil observar si los movimientos son sacados de dicho árbol observando la velocidad con la que se calculan. El árbol que Pentobi utiliza para Blokus Duo tiene tan solo 91 movimientos, lo que posibilita su implementación en hardware codificado directamente como lógica. Algunos de estos movimientos están marcados como "no deseables" – la función de los mismos es proponer una respuesta ante un adversario que los realice. A parte de este detalle, Pentobi valora todos los movimientos igual y escoge uno al azar. Para comprobar si realmente todas las opciones son igual de deseables, se procedió a simular multitud de partidas al máximo nivel posible, y se observó que Pentobi nunca perdía (contra sí mismo o contra el diseño de FPGA) con un determinado primer movimiento (ver Fig. 14), al que merece dar un peso superior.

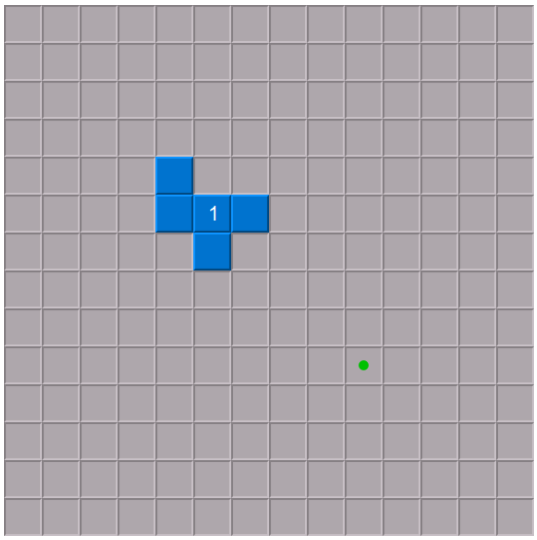


Fig. 14: Apertura fuerte

Para implementar este comportamiento, el software y hardware necesitan de un generador de números aleatorios, el único elemento no determinista del diseño – en contraste con la aleatoriedad de Monte Carlo. En nuestro caso, se puede incluso prescindir de un generador de

alta calidad y utilizar un simple contador de ciclos, eligiendo como número aleatorio el valor que tenga dicho contador cuando llegue la solicitud.

10 Conclusiones

A pesar de la complejidad exponencial del problema se ha conseguido duplicar la profundidad de exploración efectiva en muchos casos, con la consiguiente mejora de resultados en partidas. Parte de la mejora viene de un diseño hardware más rápido y más paralelo, pero la mayor parte es consecuencia de mejoras algorítmicas que reducen la cantidad de trabajo a realizar sin modificar el resultado de la exploración.

Además, se han explorado otras opciones para mejorar la inteligencia artificial sin aumentar la profundidad de exploración (que sigue estando muy limitada por carácter exponencial del problema). A tal efecto se ha desarrollado una estrategia de evaluación de nodos basada en una profundización local no exhaustiva de coste lineal, que permite obtener una visión más certera del estado de la partida a una velocidad muy superior a la de la búsqueda Minimax clásica.

Gracias al gran paralelismo a nivel de operación en el diseño hardware, éste consigue superar con creces el rendimiento del software equivalente a pesar de contar con una frecuencia de reloj 60 veces inferior. El consumo energético de la versión hardware también es una fracción muy pequeña del requerido por la versión software.

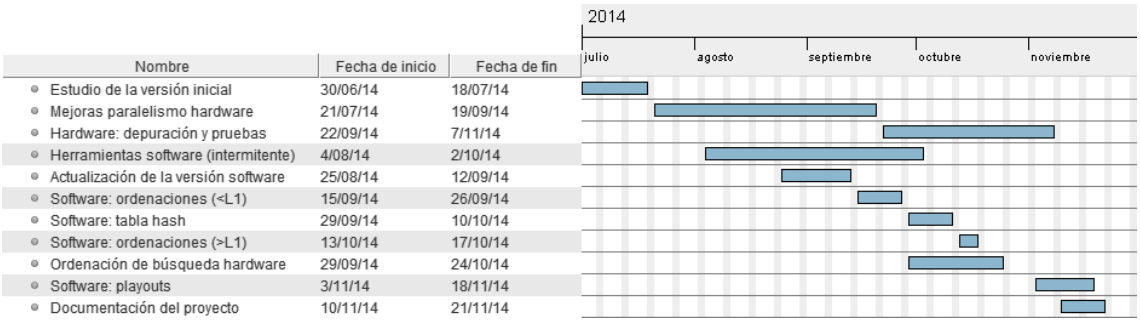
Con todo esto, el diseño jugando con un segundo de tiempo tiene aproximadamente la misma calidad que el nivel 6 del software Pentobi, una implementación que explota los puntos fuertes de las arquitecturas modernas de propósito general, utilizando varios hilos para acelerar la ejecución, una gran cantidad de memoria para dirigir la búsqueda cuidadosamente, y mucha energía para ocultar las latencias de acceso a la misma. Pentobi tiene dos niveles superiores que juegan mejor que el diseño hardware, pero estos niveles necesitan bastante más de un segundo para calcular sus movimientos por lo que la comparación no es del todo justa.

11 Trabajo futuro

El trabajo que se ha presentado no está cerrado. Durante su desarrollo han surgido diversas ideas que no se han podido desarrollar por falta de tiempo.

Una línea interesante es estudiar las nueva familias de plataformas híbridas que incluyen procesadores convencionales y FPGAs en el mismo chip. En estas plataformas podríamos mezclar módulos software con módulos hardware y tratar de buscar las combinaciones que proporcionasen mejor rendimiento y menos consumo. Dado que ya disponemos de un amplio conjunto de módulos equivalentes disponibles tanto para hardware como para software la mayor complejidad sería decidir dónde ubicar cada módulo y cómo optimizar las comunicaciones.

12 Planificación



Nombre	Fecha de inicio	Fecha de fin
Estudio de la versión inicial	30/06/14	18/07/14
Mejoras paralelismo hardware	21/07/14	19/09/14
Hardware: depuración y pruebas	22/09/14	7/11/14
Herramientas software (intermitente)	4/08/14	2/10/14
Actualización de la versión software	25/08/14	12/09/14
Software: ordenaciones (<L1)	15/09/14	26/09/14
Software: tabla hash	29/09/14	10/10/14
Software: ordenaciones (>L1)	13/10/14	17/10/14
Ordenación de búsqueda hardware	29/09/14	24/10/14
Software: playouts	3/11/14	18/11/14
Documentación del proyecto	10/11/14	21/11/14

Fig. 15: Diag. Gantt de planificación de proyecto

El proyecto ha sido desarrollado desde julio hasta noviembre de 2014, invirtiéndose primeras semanas en el estudio de la versión inicial y la familiarización con las técnicas empleadas.

Después se procedió en paralelo a implementar las mejoras hardware mientas se valoraban otras posibles técnicas y se trabajaba en las mejoras de software (tanto herramientas como la implementación de la inteligencia artificial equivalente en técnicas al hardware).

Más adelante (y en paralelo con las tareas de hardware), se comenzaron a implementar en software las nuevas técnicas de aceleración de la búsqueda, que iban añadiéndose al diseño hardware al comprobar los resultados positivos en las simulaciones software.

Por último, se investigaron las últimas mejoras de la función de evaluación basadas en *playouts*, y se redactó la documentación del proyecto.

13 Referencias

[I] ICFPT 2013 Design Competition. http://lut.eee.u-ryukyu.ac.jp/dc13/

[II] CFPT 2014 Design Contest. http://www.icfpt2014.org/Info.asp?call=1E4F09D550631DDE725F71F2CFCB0495

[III] https://en.wikipedia.org/wiki/Field_Programmable_Gate_Array

[IV] http://lut.eee.u-ryukyu.ac.jp/dc13/rules.html

[V] Russell, Stuart J.; Norvig, Peter (2003), Artificial Intelligence: A Modern Approach (2nd ed.), Upper Saddle River, New Jersey: Prentice Hall, pp. 163–171, ISBN 0-13-790395-2

[VI] https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

[VII] https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search

[VIII] http://pentobi.sourceforge.net/

[IX] https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

[x] Combining Online and Offline Knowledge in UCT. S. Gelly, D. Silver. Proceedings of the 24th international conference on Machine learning, pp. 273-280, 2007. http://www.machinelearning.org/proceedings/icml2007/papers/387.pdf

[xi] Liu, C., "Implementation of a highly scalable blokus duo solver on FPGA," Field-Programmable Technology (FPT), 2013 International Conference on , vol., no., pp.482,485, 9-11 Dec. 2013. https://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=6718423

[xii] Yoza, T.; Moriwaki, R.; Torigai, Y.; Kamikubo, Y.; Kubota, T.; Watanabe, T.; Fujimori, T.; Ito, H.; Seo, M.; Akagi, K.; Yamaji, Y.; Watanabe, M., "FPGA Blokus Duo Solver using a massively parallel architecture," Field-Programmable Technology (FPT), 2013 International Conference on , vol., no., pp.494,497, 9-11 Dec. 2013. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6718426

[xiii] https://docs.oracle.com/javase/1.5.0/docs/api/java/lang/String.html#hashCode%28%29

An improved FPGA-based specific processor for Blokus Duo

Javier Olivito, Alberto Delmás, Javier Resano
gaZ group, DIIS-I3A
University of Zaragoza
Zaragoza, Spain
jolivito@unizar.es, adelmas@gmail.com, jresano@unizar.es

***Abstract*— This article presents a hardware design of a specific processor for Blokus Duo game. This design is an evolution of our previous work presented in the ICFPT'13 Design Competition. In order to improve its performance we have designed parallel hardware blocks to speed up the most time-consuming tasks, and included additional techniques to reduce the search space. As a consequence we can process a board six times faster than in our previous version and we prune the game-tree much more efficiently.**

Keywords—Blokus Duo; FPGA; min-max; parallelism;

I. INTRODUCTION

Blokus Duo is a variant of Blokus [1] designed for two players played on a 14×14 board. Each player has 21 differently shaped tiles, and can place them with eight different rotations. Tiles can be placed only in those squares with corner-to-corner contact with a tile of the same color. Moreover, a tile cannot have edge-to-edge contact with any other tile of the same color. Each player places one tile at one time, and the game continues until both players cannot places more tiles. The score of each player is obtained taking into account the placed tiles and their size. Each placed tile adds as many points as the number of squares it occupies (from one to five). Hence, the goal is to occupy as many squares as possible while trying to reduce as much as possible the number of squares that the opponent can occupy.

This game is becoming popular and has received several international awards. It is easy to learn and very addictive. It has been selected for the ICFPT'14 Design Competition [2] following the success of last edition where 21 teams were competing [3].

However, the fact that its rules are simple does not mean it is a simple game. It is, in fact, a very complex game. On the one hand, it is difficult to know whether a player is winning or not since the score of a game can drastically change on the last movements. On the other hand, the game tree to explore is huge. For a given vertex, up to 127 different moves can be made, and during a game there are many vertices where a player can place a tile.

To evaluate a board state we use an expensive metric that takes into account not only the current score but also the regions that can be accessed by each player. We say that it is expensive because finding the squares that can be accessed by each player involves a lot of computations. But we believe that a good evaluation function is the key to develop a strong player.

Regarding the complexity of the game tree, in our previous design [4] we already identified several techniques to reduce the number of movements to explore. In this version we have added hardware support to reorder the game-tree nodes in order to increase the alpha-beta pruning efficiency, and we have also included some specific hardware to identify which are the most promising moves and discard the remaining ones.

Once we have reduced the game tree, our next goal is to process it as fast as possible. To this end we have designed several modules that take full advantage of the parallelism of the different tasks carried out by our design. For instance, all the vertices of a board are obtained in parallel in one clock cycle, as well as all the legal moves for a given vertex, and the squares that can be accessed from a vertex. These modules speed up our processor six times in comparison to our previous design.

In the following sections we briefly describe the related work, the new techniques, and the main modules of our design. Later we present some performance results, and finally we provide some conclusions.

II. RELATED WORK

Several articles describe hardware implementations of a Blokus Duo player. Reference [5] presents a co-design approach where the artificial intelligence is executed in a processor and the boards are evaluated by a hardware accelerator implemented on an FPGA. Reference [6] describes an architecture based on Monte-Carlo method that requires little resources and operates at 150MHz. References [7] and [8] present two approaches based on high-level synthesis. All of these are interesting approaches but they are not competitive against strong players. In the previous edition of the Design Competition all of them exhibit worst results than our previous design, which was awarded with the fourth prize. The winners described their design in [9]. They implemented forty evaluators in parallel leading to a great performance. The performance of our previous design was worst, but our design used much less resources (27% of a Xilinx Spartan-6 LX45), and the winners used 93% of an Altera Arria II GX125, which is a larger FPGA. In any case, since our current

design is clearly stronger than our previous one, which was already competitive, we believe that we are presenting a state-of-the-art design.

III. TECHNIQUES

Our design generates a game tree in a depth-first fashion and searches for the best move following the min-max algorithm with alpha-beta pruning. The branching factor is very large in this game, so additional pruning techniques are required in order to look ahead as much as possible. To this end, we have implemented the following techniques:

A. *Node reordering*

The efficiency of alpha-beta pruning strongly depends on the order the nodes are explored. The sooner the best nodes are explored, the higher the pruning efficiency is. In order to take advantage of this property, we first create a small game tree that only analyses our movements and the opponent responses to those movements, in such a way that we get an estimation of which movements are the most promising. This game tree is generated without alpha-beta pruning since it would degrade the quality of estimations because pruned movements are upper bound scored. The outcome of this preliminary exploration is stored in a small memory, and in deeper explorations we generate the game tree following this arrangement.

B. *Move discard*

Players must compete for shared areas. Placing a tile in an area where the opponent does not have access is usually a bad idea. We propose the following algorithm to decide which movements can be discarded.

```
ALGORITHM move_discard
  create_overlapping_map;
  for each vertex do
    if vertex in overlapping map then
      for each movement in vertex do
        if tile_overlapping ≥ overlapping_threshold then
          explore_move;
        end if
      end for
    end if
  end for
```

The overlapping threshold varies during a game. At the beginning the movements are required to be highly overlapped, and as the game progresses this requirement is progressively relaxed. The overlapping map identifies the squares that are accessible by our opponent or that are adjacent to its tiles. Fig. 1 depicts the overlapping map at the beginning of a game. In this case three of the seven blue vertices are discarded since they are not in the relevant area.

C. *Tile size*

Areas in dispute are especially relevant in the first movements. Placing big tiles allows the player to reach farther areas as soon as possible. Moreover, big tiles are easier to place when the board is mostly empty. Hence, early in the game only big tiles should be considered.

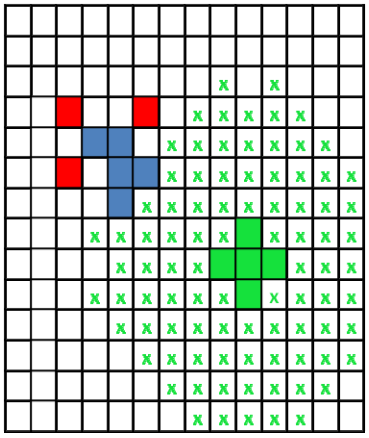


Fig. 1. Overlapping map. Movements in vertices marked in red are discarded

IV. MAIN MODULES

From an architectural point of view, our design has been focused on exploiting board and piece parallelism. This enables large speedups in critical tasks like legal-moves search or board evaluation.

A. *Board*

We consider interesting not to store only the player who occupies each square (Free, Hero, Rival), but also which players can place new tiles on them (Free, Forbidden Hero, Forbidden Rival, Forbidden Both).

This idea reduces the number of squares to analyze when looking for legal moves or evaluating a board.

B. *Move Writer*

This module is responsible for updating the board after each tile is placed. Write operations are done piece by piece, with a latency of *n* cycles where *n* is the size of the tile to be written. Every cycle it writes a piece of tile (marking it as forbidden for both players), and it updates its north, south, west and east neighbors.

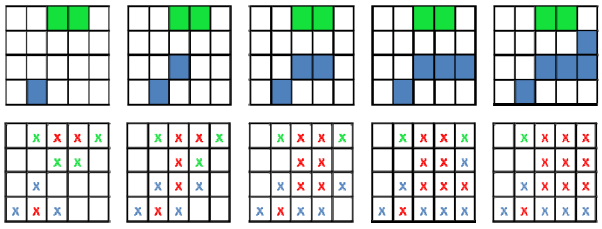


Fig. 2. Example of move writing placing tile 'f'.

Fig. 2 illustrates the operation with an example where the blue player places the ‘f’ tile. The leftmost board is the initial state, and the rightmost four represent its evolution during the four cycles required to write the movement. The boards on the bottom represent which player can place in each square. A blue ‘x’ means that blue player cannot place there; a green ‘x’ means that green player cannot place there, and a red ‘x’ means that none of them can place there.

C. *Vertices Map*

Every operation on a board is done on a vertex-by-vertex basis. We have implemented a module which identifies all the valid vertices of a given board. It is a matrix of combinational modules, where each module computes a given vertex. To this end it analyses its four surrounding squares looking for one of the four vertex patterns shown in Fig. 3. This module has been replicated to analyze the board in parallel for both players.

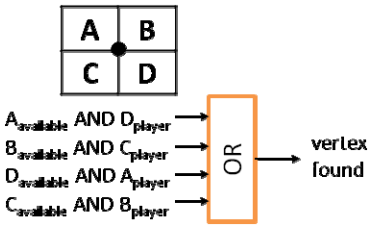


Fig. 3. Vertex detector.

D. *Processing Window*

When we are exploring a vertex, we are interested in every potentially reachable square from that vertex. Given the shapes of the tiles and the possible rotations, the region of interest has a shape such as the one shown in Fig. 4. Therefore, we have implemented a module mainly composed of multiplexers which provides this information around any square. We use it to identify the legal moves and to compute the accessibility. As the vertices map, this module is replicated in order evaluate the board in parallel for both players.

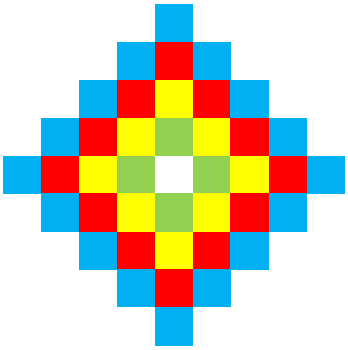


Fig. 4. Processing Window shape

E. *Legal Move Finder*

Finding the possible moves of a board is a critical task since it has to be performed for every node in the game-tree. We have implemented a module that provides one legal move per cycle.

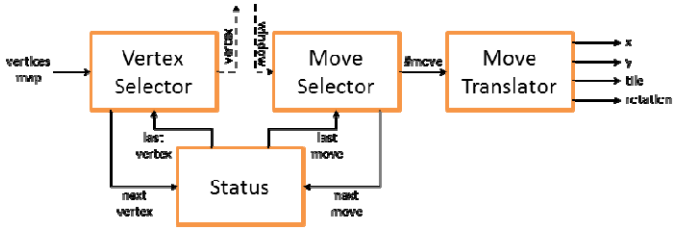


Fig. 5. Legal Move Finder

The Vertex Selector (shown in Fig. 5) receives all the identified vertices and returns the next vertex to explore. It consists of a mask that hides the vertices already explored, and a priority encoder that selects the next unexplored one. Once this next vertex is selected, the processing window is centered on it, and the Move Selector (also shown in Fig. 5) tests all the potential moves in parallel. Then we follow the same scheme as the Vertex Selector, selecting one move per cycle. Finally, the Move Translator decodes the selected move providing the needed information (x, y, tile and rotation).

F. *Accessibility Evaluator*

In this game it is desirable to reach as much area as possible, and to reduce the area reachable by our opponent. Hence to evaluate a board we first analyze the area that can be reached by each player. Our accessibility evaluator processes all the vertices identified by both Vertices Maps. It integrates a Vertex Processor (see Fig. 6), which is a combinational block that identifies the reachable squares from a vertex in one clock cycle. For each vertex, the Vertex Processor identifies the reachable squares taking into account the board status and the available tiles, resulting in a 14x14 map indicating which squares are accessible. Finally, a Tree Adder returns the accessibility value for each player (i.e. the number of squares that can be accessed by each player).

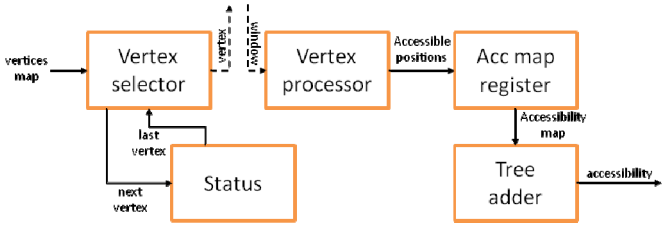


Fig. 6. Accesibility Evaluator

G. *Node Reordering*

This module receives the explored move candidates and their score, and sorts them storing the best scoring one in position 0. Fig. 7 depicts the data path of this module. It consists of a memory that stores the movements sorted by their scores (Sort RAM); a register to latch new movements to be sorted (New move reg); a counter which points to the first empty position (First empty counter); an additional counter to carry out the sort process (Sort counter); and a comparator which identifies if the position of two movements needs to be interchanged. Every time a new movement is evaluated, it is latched in the *New move register* and the address of the last movement stored in the *Sort RAM* is loaded in the *Sort Counter*. Then the scores of the new movement and the previously stored are compared; if the new movement is better, their positions are interchanged.

It takes two cycles to complete each interchange, but this latency is fully hidden within the tree search.

The efficiency of this technique strongly depends on the game scenario. We have observed situations where it allows our processor to reduce the search space by 90%.

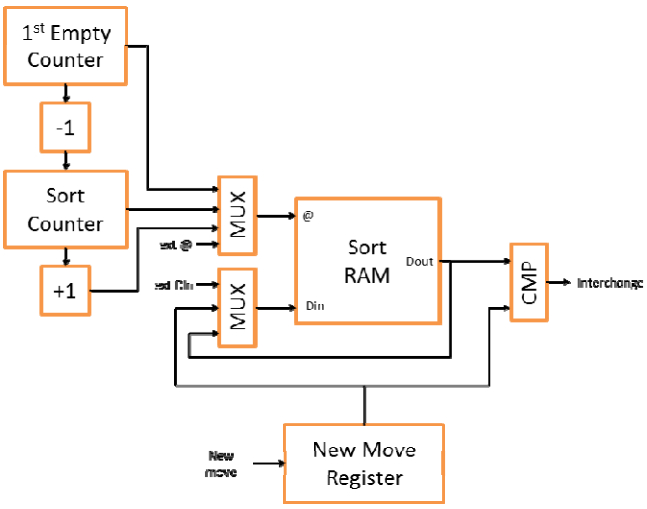


Fig. 7. Node Reordering

V. RESULTS

Our design has been implemented on a Xilinx Virtex-5 LX110T. It is able to process, on average, about 1.2M boards per second. Table I shows the performance results of the most relevant tasks to decide a move. The next move to process is found in one clock cycle; afterwards, up to five cycles are needed to generate the new board; and finally one clock cycle per vertex to evaluate the board. Column 'Time' is calculated for 50 MHz, which is the frequency our design can run in this FPGA.

TABLE I. TASKS PERFORMANCE

Task	Cycles	Time (ns)
Find next move	1	20
Generate a new board	*tile_size*	*tile_size*×20
Evaluate a board	*#vertices*	*#vertices*×20

Our design presented in ICFPT '13 was able to process about 0.2M boards per second. It means that we have achieved a speedup of 6.

In order to test our processor against a strong opponent, we played several games against Pentobi [10], which is currently the strongest Blokus software. The latest version of Pentobi has eight levels and it is able to decide every movement in less than one second up to level five. Levels six, seven and eight require up to tens of seconds. Our design decides every movement in no more than one second. Table II depicts these results. We have played ten games in each level, five as blue and five as green. The results show that our design deciding the move with a one second timeout is as competitive as Pentobi in level five. Moreover, in some cases it is able to defeat Pentobi in its highest levels. Hence, with the same time budget, our design is roughly as strong as Pentobi.

TABLE II. RESULTS AGAINST PENTOBI

Level	1	2	3	4	5	6	7	8
Results (Win-Lost)	10-0	10-0	10-0	8-2	6-4	3-7	1-9	1-9

Table III shows the resource utilization for a design with support for a game-tree of up to ten levels.

TABLE III. FPGA RESOURCES UTILIZATION

Slice Registers	Slice LUTs	BRAMs
5,959 (8%)	19,413 (28%)	12 (8%)

VI. CONCLUSIONS

With our previous design we demonstrated that is was possible to develop a competitive hardware design for a complex problem in just three months. However, we left many optimization opportunities unexplored due to the time constraints. In this design we have taken advantage of task parallelism. As a result our new design is six times faster. Moreover, we have included interesting optimizations to prune the game-tree more efficiently. The results demonstrate that we are competitive against a strong software application as Pentobi.

Acknowledgment

This work was supported in part by grants TIN2010-21291-C02-01 (Spanish Gov. and European ERDF), gaZ: T48 research group (Aragón Gov. and European ESF), and HiPEAC-3 NoE (European FET FP7/ICT 287759).

References

[1] Blokus. http://en.wikipedia.org/wiki/Blokus

[2] International Conference on Field-Programmable Technology 2014 http://www.icfpt2014.org

[3] International Conference on Field-Programmable Technology 2013 http://www.fpt2013.org

[4] J. Olivito, C. Gonzalez, and J. Resano, "An FPGA-based specific processor for Blokus Duo", International Conference on Field-Programmable Technology 2013. Kyoto, Japan. pp. 502-505.

[5] Kojima, A, "An implementation of Blokus Duo player on FPGA," International Conference on Field-Programmable Technology 2013. Kyoto, Japan. pp.506-509.

[6] Liu, C., "Implementation of a highly scalable blokus duo solver on FPGA," International Conference on Field-Programmable Technology 2013. Kyoto, Japan. pp.482-485.

[7] Sugimoto, N.; Miyajima, T.; Kuhara, T.; Katuta, Y.; Mitsuichi, T.; Amano, H., "Artificial intelligence of Blokus Duo on FPGA using Cyber Work Bench," International Conference on Field-Programmable Technology 2013. Kyoto, Japan. pp.498-501.

[8] Jiu Cheng Cai, *et al.*, "From C to Blokus Duo with LegUp high-level synthesis," International Conference on Field-Programmable Technology 2013. Kyoto, Japan. pp.486-489.

[9] T. Yoza, *et al.* "FPGA Blokus Duo Solver using a massively parallel architecture", International Conference on Field-Programmable Technology 2013. Kyoto, Japan. pp. 494-497.

[10] Pentobi webpage. http://pentobi.sourceforge.net/