Solving Parity Games on the Playstation 3

Freark van der Berg
University of Twente, The Netherlands
f.i.vanderberg@student.utwente.nl

ABSTRACT

Parity games are a type of game in which two players 'play' on a directed graph. Solving parity games is equivalent to model checking for μ-calculus. Thus, parity game solvers can be used for model checking. This requires a lot of computational power. Many-core CPUs generally have much more computational power than other CPUs. The Playstation 3 contains an advanced, modern many-core CPU, the IBM Cell Broadband Engine Architecture (CBEA). It is a low-cost option to investigate developing efficient algorithms for many-core CPUs. However, developing efficient algorithms for The Cell remains largely uncharted territory. The Small Progress Measures parity games algorithm, developed by Marcin Jurdziński, is poised for running on the IBM Cell Broadband Architecture, in particular for the Playstation 3. Here we show there are six important aspects regarding optimizing this algorithm for the Playstation 3: graph subdivision, lifting order, cluster issuing order, capturing cycles, the lifting heuristic and preprocessing the graph. Various optimizations are proposed, e.g.: cluster regeneration, altivec instruction set implementation, and the cluster dependency heuristic. Some of these were implemented and benchmarked. The benchmarks indicate the Playstation 3 seems a viable architecture for the Small Progress Measures algorithm.

Keywords

Playstation 3, IBM Cell Broadband Engine Architecture, Small Progress Measures, Parity Game, Model Checking, Algorithm Optimization, Benchmarking

1. INTRODUCTION

Parity games are a type of game in which two players 'play' on a directed graph. Solving parity games is equivalent to model checking for μ-calculus [14]. Thus, algorithms for solving parity games can be used for model checking, which makes parity games an interesting subject. One of the algorithms for solving parity games is the Small Progress Measures algorithm, developed by Marcin Jurdziński [8]. A parallelized version of this algorithm was implemented by Jaco van de Pol and Michael Weber, for the x86 architecture [12]. Jorne Kandziora made an implementation for the Cell Broadband Engine Architecture, based on the x86 version [6]. This implementation was not complete, unoptimized and unbenchmarked.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
13th *Twente Student Conference on IT* June 21st, 2010, Enschede, The Netherlands.
Copyright 2010, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

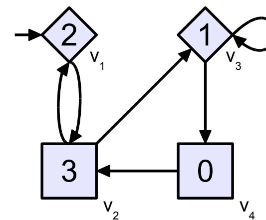

Figure 1. Parity Game

This research continued with the implementation made by Kandziora. The reason for this follow-up research lies with the architecture the algorithm is aimed for. Many-core CPUs provide a lot of computational power, reaching even 1 TFLOPS [7]. The Cell CPU in the Playstation 3 is a good example of a modern multi-core CPU, since it contains a single general-purpose processor and eight co-processors [6]. This is similar to many-core CPUs. A big difference is the price. The Playstation 3 is a lot cheaper[1] and therefore a low-cost solution. Moreover, there is a proper Software Development Kit available [4], aiding the development. The CBEA is still relatively young and developing efficient algorithms for it has not yet been widely researched. This paper indirectly also investigates the possibilities of the Cell CPU as a target architecture for other algorithms.

This section introduces parity games in general, the Small progress Measures algorithm, and the Cell Broadband Engine Architecture. Section 2 describes the goal of this research. Section 3 lists work related to this research. Section 4 describes possible optimizations. Section 5 discusses an implementation of a few optimizations. Section 6 shows benchmarks of these optimizations. Section 7 contains the conclusions of this research. Section 8 suggests possible future research.

1.1 Parity Games

A parity game is played on a directed graph. Each vertex has a number, called a priority. Starting from the starting vertex, player Even and player Odd take turns moving a token along the edged of the graph. The result is a path, called the play. Winning is different for finite plays and infinite plays. In a finite play, a player wins when the opponent cannot make a move. In an infinite play, the winning player is determined using the priorities of the

[1]Today a PS3 costs around $300, while for example the Ambric Am2045 chip cost $325.000 per thousand units [2] in 2006.

Table 1. Plays in the example

Path	Play	Winner
$v_1, v_2, v_1, v_2, ...$	2,3,2,3,...	Even ($\diamond$)
$v_1, v_2, v_3, v_3, ...$	2,3,1,1,...	Odd ($\Box$)
$v_1, v_2, v_3, v_4, v_2, v_3, v_4, ...$	2,3,1,0,3,1,0,...	Even

vertices. Player Even wins if the smallest priority that occurs infinitely often in the play is even. Player Odd wins if it is odd. For this research only infinite plays are regarded.

Consider the graph pictured in Figure 1. This is a very small example, as parity games in general can easily contain millions of vertices.

It will be player Even's turn when the token is on a diamond and player Odd's turn when the token is on a square. The possible plays in this simple example are listed in Table 1. Note that there is a path where Player Odd can win. However, this depends on the choice of Player Even. So if Player Even uses a proper *strategy*, Player Odd will never win.

Formally, a parity game is described as $G = (V_\Box, V_\diamond, E, p)$, where

- $V_\diamond$ is the finite set containing the vertices on which player Even moves;
- $V_\Box$ is the finite set containing the vertices on which player Odd moves;
- vertices $V := V_\Box \uplus V_\diamond$;
- edges $E \subseteq V \times V$;
- priority function $p : V \rightarrow \{i \mid 0 \leq i < d\}$, d the number of priorities

1.2 Small Progress Measures

The Small Progress Measures algorithm was devised by Marcin Jurdziński [8]. A short summary of the algorithm is as follows. The Small Progress Measures algorithm assigns a vector to each vertex, assigning n vectors in total. These vectors have length d and initially they are $\vec{0}$. The following is assumed:

$$\text{priority function } p : V \rightarrow \{i \mid 0 \leq i < d\}$$

If it does not hold, the priorities can be rearranged so it does. Let m_i be the number of vertices with priority i:

$$m_i := |\{v \in V | p(v) = i\}|, 0 \leq i < d$$

Define M_i as:

$$M_i := \begin{cases} \{0\} & , i \text{ even} \\ \{0, ..., m_i\} & , i \text{ odd} \end{cases} , 0 \leq i < d$$

Then measures M_G contains all possible vector configurations:

$$\begin{aligned} M_G &:= M_0 \times M1 \times ... \times M_{d-1} \\ M_G^\top &:= M_G \cup \{\top\} \end{aligned}$$

Note that all the even components are always equal to 0. The top element $\top$ is defined as:

$$\forall \alpha \in M_G : \alpha < \top$$

Lexicographical ordering of vectors $\alpha, \beta \in M_G$ is denoted by:

$$\alpha, \beta \in M_G : \alpha < \beta$$

Ordering using up to the ith component is denoted by:

$$<_i, =_i, >_i\text{: compare up to the } i^{\text{th}} \textit{component}$$

For the algorithm, we leave out the priorities in even components, e.g.:

$$\langle x, y \rangle = (0, x, 0, y, 0)$$

Furthermore, we define the successors of vector α as:

$$\begin{aligned} \text{succ}_i(\alpha) : \quad & \text{minimal } \beta \in M_G^\top, \text{such that} \\ & \alpha = \beta = \top \text{ or} \\ & \alpha \leq_i \beta \text{ even } i \\ & \alpha <_i \beta \text{ odd } i. \end{aligned}$$

The goal of the algorithm is to compute the assignment $\varrho : V \rightarrow M_G^\top$, such that $\forall v \in V$:

- $v \in V_\diamond : \exists (v, w) \in E : \varrho(v) \geq_{p(v)} \text{succ}_{p(v)}(\varrho(w))$
- $v \in V_\Box : \forall (v, w) \in E : \varrho(v) \geq_{p(v)} \text{succ}_{p(v)}(\varrho(w))$

The complete algorithm is as follows:

$\varrho := \lambda v \in V.(0, ..., 0)$
while $\exists U \subseteq V \cdot \varrho < \textit{Lift}(\varrho, U)$ **do** $\varrho := \textit{Lift}(\varrho, U)$

$$\textit{Lift}(\varrho, U)(v) = \begin{cases} \varrho(v) & , v \notin U \\ \min_{(v,w) \in E} \textit{succ}_{p(v)}(\varrho(w)) & , v \in U \cap V_\diamond \\ \max_{(v,w) \in E} \textit{succ}_{p(v)}(\varrho(w)) & , v \in U \cap V_\Box \end{cases}$$

From this can be seen that the lifting of a vertex depends on its children. So changes always propagate backwards through the graph. If a vertex changes, its parents need updating.

The result is then given by Jurdziński:

THEOREM(JURDZIŃSKI [8]) 1. *The winning set* $W_\diamond = \{w \mid \varrho(w) \neq \top\}$. *The strategy* $\tilde{\varrho}(v : V_\diamond) := \min_{(v,w) \in E}(\varrho(w))$ *is a winning strategy for player Even.*

The algorithm has time complexity $O(m \times n^{\lfloor d/2 \rfloor})$ and needs $O(d \times n)$ space [8].

1.3 SPM algorithm applied

The Small Progress Measures algorithm can be applied to the parity game graph shown in Figure 1. As $d = 4$ and $m_i = 1, 0 \leq i < d$:

$$\begin{aligned} M_0 &= \{0\} \\ M_1 &= \{0, 1\} \\ M_2 &= \{0\} \\ M_3 &= \{0, 1\} \end{aligned}$$

This results in:

$$\begin{aligned} M_G &= \{0\} \times \{0, 1\} \times \{0\} \times \{0, 1\} \\ M_G^\top &= M_G \cup \{\top\} \end{aligned}$$

We then start lifting until a fix point is reached. Lifting can be done on any vertex, so we start at v_4:

$$v_4 \rightarrow v_2 : \text{succ}_0\langle 0, 0 \rangle = \langle 0, 0 \rangle$$

This lift yielded no change. We continue with v_3, v_2 and v_1.

$$\begin{aligned} v_3 \rightarrow v_3 : \text{succ}_1\langle 0, 0 \rangle = \langle 1, 0 \rangle \\ v_3 \rightarrow v_4 : \text{succ}_1\langle 0, 0 \rangle = \langle 1, 0 \rangle \end{aligned}$$

So we update the measure of v_3 with $\min\{\langle 1, 0 \rangle, \langle 1, 0 \rangle\} = \langle 1, 0 \rangle$.

$$\begin{aligned} v_2 \rightarrow v_1 : \text{succ}_3\langle 0, 0 \rangle = \langle 0, 1 \rangle \\ v_2 \rightarrow v_3 : \text{succ}_3\langle 1, 0 \rangle = \langle 1, 1 \rangle \end{aligned}$$

So we update the measure of v_2 with $\max\{\langle 0,1\rangle, \langle 1,1\rangle\} = \langle 1,1\rangle$.

$$v_1 \to v_2 : \text{succ}_2\langle 1,1\rangle = \langle 1,0\rangle$$

So we update the measure of v_1 with $\langle 1,0\rangle$.

All the steps are listed in the following table. Note that the last iteration over all the vertices (the last four steps) does not yield any change, which means we have reached the fix point.

Step	v_1	v_2	v_3	v_4	Comment
0	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$	Initial
1	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 0,0\rangle$	Lifting v_4
2	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle 1,0\rangle$	$\langle 0,0\rangle$	Lifting v_3
3	$\langle 0,0\rangle$	$\langle 1,1\rangle$	$\langle 1,0\rangle$	$\langle 0,0\rangle$	Lifting v_2
4	$\langle 1,0\rangle$	$\langle 1,1\rangle$	$\langle 1,0\rangle$	$\langle 0,0\rangle$	Lifting v_1
5	$\langle 1,0\rangle$	$\langle 1,1\rangle$	$\langle 1,0\rangle$	$\langle 0,0\rangle$	Lifting v_4
6	$\langle 1,0\rangle$	$\langle 1,1\rangle$	$\langle 1,0\rangle$	$\langle 0,0\rangle$	Lifting v_3
7	$\langle 1,0\rangle$	$\langle 1,1\rangle$	$\langle 1,0\rangle$	$\langle 0,0\rangle$	Lifting v_2
8	$\langle 1,0\rangle$	$\langle 1,1\rangle$	$\langle 1,0\rangle$	$\langle 0,0\rangle$	Lifting v_1

Now we use Theorem 1 to obtain the winning set:

$$W_\diamond = \{w | \varrho(w) \neq \top\} = V$$

This means player Even ($\diamond$) has a winning strategy starting from any vertex.

1.4 Cell

The Cell Broadband Engine Architecture features a wide range of combinations of various processing elements, but for this research only the configuration in the Playstation 3 is regarded. This configuration contains a main processor called the "PowerPC processor element" (PPE), eight co-processors called "Synergistic Processor Element"s (SPE), a memory flow controller (MFC), and the interrupt controller (IIC). An SPE contains a "Synergistic Processor Unit" (SPU) with dedicated local storage of 256kB, a dedicated MFC with a memory management unit (MMU) and a replacement management table (RMT). Communication between elements is done via an element interconnect bus (EIB). Figure 2 shows this graphically, also showing memory and I/O units [6].

The Cell is not the only possible architecture to consider. Graphics cards are also powerful. However, they have two important limitations. First, the local storage of the GPU cores is much too small. Second, the GPU itself usually lacks the possibility of branching and if it does support branching, it comes with a severe performance penalty. However, modern graphics cards have improved on this. The local storage has increased [11], but still is not up to par with the Cell. Branching has also been improved [11].

The amount of local storage of each processing unit in a graphics card is very low. This means access to the main memory will be done a lot more frequent, reading subgraphs and writing results. This makes memory bandwidth a major bottleneck. The memory bandwidth of the latest graphics cards even surpasses[2] the Cell, but not enough to compensate. So until graphics cards have more local storage per processing unit, they are not an option. This applies to the Small Progress Measures algorithm, but other model checking techniques could benefit from graphics cards [1].

[2]The Cell's memory bandwidth when clocked at 3.2GHz is 204.8GB/s [6]. The nVidia GTX 295's memory bandwidth is 223.8GB/s [10].

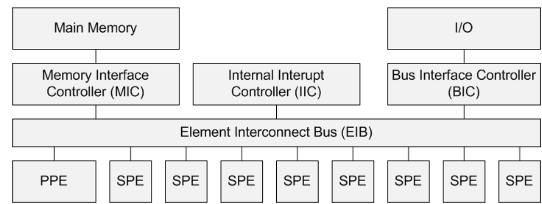

Figure 2. Cell CPU architecture in the PS3

2. PROBLEM STATEMENT

This paper investigates i) where the performance bottlenecks lie when running this algorithm on The Cell CPU; ii) what optimizations are possible under which constraints; iii) and how much is gained from these optimizations. The primary targets for optimization are i) the heuristics to divide the load into pieces; ii) the so-called lifting operation; iii) and platform specific instructions. Dividing the load makes parallel processing possible. The lifting operation is a key element of the Small Progress Measures algorithm. Platform specific instructions are for example the *altivec instructions* [5]. These instructions provide an improvement over regular instructions for vector operations.

2.1 Memory Access

In the x86 version of the algorithm, the implementation is based on uniformly accessible shared memory. In the Cell CPU, this assumption cannot be made. SPEs each have their own fast local storage, only 256kB, and slower access to the main memory has to go via the EIB. Since the Small Progress Measures algorithm runs in $O(n \times d)$ space and a parity game graph can easily contain millions of vertices, the main memory is certainly needed. Slow memory access can be minimized by subdividing the large graph into smaller graphs, called *clusters*. A cluster can then be loaded into the small storage of an SPU. After performing the algorithm on the cluster, the result will be written back to the combined result in the main memory. Jurdziński shows that this yields the same result as performing the algorithm on the entire graph [8]. This way, access to the cluster can be done locally. However, access to measures of vertices outside the cluster is not local, so the problem is not resolved entirely. The graph subdivision should take this issue into account.

2.2 Graph Subdivision

Subdividing a large graph into clusters is an important part of the algorithm. For the Cell implementation it may even be the most important. A cluster cannot be too large, because of the local storage limit of 256kB. Too small clusters will result in relatively more overhead because of the memory transfers. The goal is a maximally sized cluster, a so-called *full cluster*, when no more can be fitted in the local storage of an SPU.

Moreover, the number of SPUs in use is also important. Intuitively, we say that the more enabled SPUs, the faster the algorithm. However, more SPUs causes more memory access, taxing the EIB even more. So the exact influence of the number of SPUs is not so obvious.

The method used to select clusters is a key element of subdivision. It is essential to minimize the memory access from an SPU to the main memory. Selecting clusters in such a way that they have no outgoing edges is not feasible, because many of these clusters will be much too large for the 256kB of local storage. Selecting clusters that have outgoing edges results in obtaining data from the slower main memory.

An important issue is the handling of cycles. In parity games, cycles tend to cost a lot of calculation, because changes propagate over and over within a cycle. To this end, the ideal solution would be to capture the cycle in a single cluster. This would mean the many lifts will be performed as fast as possible on a single SPU. If a cycle is in multiple clusters, the changes propagate much slower on cluster boundaries, because multiple SPUs need to wait for each other.

2.3 Lifting Order

The choice of vertices to perform the lift operation on is important. On the same graph, a bad choice of sequence could result in much worse performance [12]. Because vertices can be lifted in any order, strategies may be devised optimizing the order of lifting.

3. RELATED WORK

To subdivide the parity game graph into smaller graphs, Kandziora described a k-bounded depth-first search algorithm [9]. This algorithm starts at a vertex and defines a new cluster by adding all vertices up to a maximum depth of k or until the cluster is full. If the maximum depth is reached and the cluster is not full, the DFS search will resume at a new starting node. This algorithm is used for the clustering.

A heuristic to choose which vertices to perform the lifting on is the *focus list*, described by van de Pol and Weber [12]. This heuristic basically entails that lifted vertices will have priority over other vertices, when choosing which vertices to lift. They mention other heuristics, but these do not seem feasible on the Cell CPU, due to the small local storage of each SPE. One is the *swiping* heuristic. Due to its simplicity, the swiping heuristic is used.

Other model-checking techniques have also been explored, for example using graphics cards with general purpose graphic processing units (GPGPUs). Bošnački, Edelkamp and Sulewski used probabilistic model checking, because this relies on matrix vector multiplication [1]. A GPU can perform this kind of calculations very fast, making the graphics card an ideal architecture. They confirmed this with case studies, observing a significant speedup over standard CPU implementations. While they have investigated model-checking using a GPU, this research investigates the possibility of model-checking on the PS3, using the Small Progress Measures algorithm.

4. OPTIMIZATIONS

Optimizing the Small Progress Measures algorithm for the Cell CPU is separated into multiple parts. Capturing cycles is a part of the graph division, but because of its importance it has its own section.

1. The heuristic for dividing the graph into clusters;
2. The heuristic for selecting the order of lifting the vertices inside a cluster;
3. The heuristic for selecting the order of issuing clusters to SPUs;
4. Capturing cycles in a single cluster;
5. The lifting of individual vertices;
6. Preprocessing the graph.

Some of the proposed optimizations are not tested and are only listed as possible optimizations. Future research may benefit from these ideas or inspire new ideas.

An issue that is important when optimizing an algorithm is keeping track of the validity of the results. To this end, the results will be compared with an implementation for the x86 architecture made by Verver. This implementation has gone through and passed extensive testing [13] and is therefore assumed to be correct.

4.1 Graph Subdivision

Dividing a graph can be done in various ways. The question is what subdivision is beneficial to the Small Progress Measure algorithm. We know that the lifting of vertices depend on their children. Therefore, a depth-first search would be an idea, since children will be lifted before their parents, providing a free, convenient lifting order. Because the SPUs can contain only so many vertices, the depth-first search will have to be bounded by a certain depth.

Clustering the graph is done on the main processor, the PPU. This means not all computational power is used for the clustering. Thus, this part is a bottleneck in the current implementation, because the cluster issuing starts after the whole graphs is clustered. An obvious optimization would be to start issuing clusters as soon as they are ready.

4.2 Lifting order

By optimizing the lifting order, unnecessary lifting of vertices can be avoided. A vertex is lifted unnecessarily if the vertex is lifted without yielding any change. This comes into play on the SPU when a cluster has been issued. Various heuristics can be used for this:

1. *Swiping* [12] is a heuristic that lazily iterates through the list of vertices, front to back or back to front. This is not a very good heuristic, because a lot of vertices are unnecessary lifted.
2. *Focus List* [12] is a heuristic that prioritizes lifting vertices which were lifted the most. This way the focus shifts towards the vertices needing the most attention. A lot of unnecessary lifts are avoided using this heuristic.
3. *MaxMeasures* [13] is a heuristic that prioritizes lifting vertices with the highest measures. Research has shown this is a good heuristic.

For this research, only the swiping heuristic was implemented, because of its simplicity.

4.3 Cluster order

Like lifting of vertices, clusters can also be lifted unnecessarily. This is the case when lifting a cluster yields no change. The same sort of heuristics can be applied as well:

1. *Swiping* isn't a good heuristic in this situation either. A lot of clusters are lifted unnecessarily.
2. *Focus List* is a better option here as well. A lot of unnecessary lifts are avoided by focusing on the clusters that are lifted the most.
3. *Cluster dependency* is a technique using the dependency clusters have on each other. A cluster depends on another cluster if there's a vertex in the first cluster depending on a vertex from the second cluster. Because change propagates backwards at the vertex level, change also propagates backwards at the cluster level. This means that if lifting a cluster yielded change, its parent clusters need updating. Similarly,

if lifting children clusters of a certain vertex yields no change, the vertex does not need updating. This can be used to avoid unnecessary lifting of clusters. It is questionable whether this would yield better results than the focus list, because both try to emphasize issuing clusters needing the most lifts. However, the focus list doesn't require the overhead of keeping track of dependencies.

For this research, only the swiping heuristic was implemented, because of its simplicity.

4.4 Capturing cycles

To capture a cycle so that it ends up on a single SPU is a task faced by two problems. First, a cycle can be too big for a single SPU for which there is no solution. Secondly it can take a lot of time to localize the cycles and make use of this information. Because the best known algorithm [3] is not distributive, only the PPU would be used. The algorithm is $O(n)$, however the n can be in the millions.

To capture a cycle in a single cluster, we can use an optimistic approach. If after the clustering the cycle was caught in a single cluster, then the cycle will be lifted optimally with regard to the cycle. If it was not caught, it will take many iterations of issuing clusters before the cycle is lifted. To avoid this, after a certain number of iterations, say 20 or so, we can partially regenerate the clusters for the vertices that still need lifting. This could capture the cycle. Then we continue the iterations of issuing clusters to the SPUs. If the cycle was caught, the cycle will be lifted optimally with regard to the cycle on a single SPU, else we try regenerating the clusters again after a certain number of iterations.

The result depends on how we regenerate the clusters. If we use the same technique as the last clustering, we can possibly get the same clusters. To avoid this, various differentiations can be made:

1. Clustering technique can be changed, but this research only regards the k-bounded depth-first search.
2. The k-bounded depth-first search can be differentiated on the value of k and the starting vertex. To differentiate the starting vertex, there are various possibilities. For example:
   (a) Go through the list of vertices front to back;
   (b) Go through the list of vertices back to front;
   (c) Pick a random vertex from the list of vertices.

Though detecting every cycle beforehand can be costly, it is another option. This is only in the case that all cycles are small enough to fit in the clusters or only small enough cycles are considered. Note that one can also consider only bipartite graphs, where these cycles do not play a role. This is because on these graphs, players move in a strictly alternating manner.

4.5 Lifting vertices

This part is where hardware-specific optimizations come into play. The algorithm doesn't allow much optimization in this part, because of its apparent simplicity. Lifting a vertex entails finding a maximum or a minimum of a vector and finding a successor of this vector. The SPU has a partly implemented altivec instruction set [5], aimed at speeding up vector operations. The measures in the algorithm are implemented as a single vector of four integers, limiting the maximum priority to 8. This enables the possibility of using the altivec instruction set.

4.6 Graph Preprocessing

This preprocessing functionality was partly borrowed from Weber and Van de Pol [12] and Verver [13]. Their implementation was adapted to the implementation for the Playstation 3. This preprocessing removes self-loops, because they take a lot of unnecessary time. For these kind of self-loops, the resulting measure is known beforehand: it is either the 0-vector or the $\top$-vector. If the self-loop resulting in the $\top$-vector is not removed, the number of lifts needed would be about the product of the components of the $\top$-vector. The benchmarks illustrate this further.

A further improvement on this is to consider cycles of more than one vertex. These cycles are detected as part of the k-bounded clustering, so currently only cycles within a single cluster are considered.

5. IMPLEMENTATION

Some of the optimizations proposed in section 4 were implemented and benchmarked. This section provides the means to replicate the majority of the implementation.

5.1 Graph Subdivision

The following pseudocode describes how we can generate a cluster using a k-bounded depth first search, while preprocessing some cycles. It keeps track of the local vertex ID of every vertex. A local vertex ID is the vertex ID a vertex has in its cluster. It also keeps track of in which cluster a vertex is in. This is later used to build cluster packages, ready to be issued to an SPU. The `handle_cycle()` function is discussed in section 5.6. This function is called multiple times, as long as there are vertices to be clustered.

```
1  vector vertexInCluster;
2  vector vertexIDInCluster;
3
4  generate_cluster( cluster CL, int size,
5                    int depth) {
6    vector worklist;
7    while(CL.size < size) {
8      vertex_id V = get_next_starting_vertex();
9      worklist.push_back(V);
10     CL.vertexIDs.add(V);
11     vertexInCluster[V]   = CL.id;
12     vertexIDInCluster[V]= CL.size++;
13     V.clustered = true;
14     while(!worklist.empty() &&
15           CL.size < size) {
16       vertex_id S = worklist.back();
17       if(worklist.size() < depth) {
18         vertex_id C = get_next_free_child(S);
19         if(!S.clustered) {
20           if(C) {
21             worklist.push_back(C);
22             CL.vertexIDs.add(C);
23             vertexInCluster[C]   =CL.id;
24             vertexIDInCluster[C]=CL.size++;
25             C.clustered = true;
26           } else {
27             worklist.pop_back();
28           }
29         } else if(vertexInCluster[C]==CL.id) {
30           handle_cycle(C, cluster_id , worklist);
31         }
32       } else {
33         worklist.pop_back();
34       }
35     }
36   }
37 }
```

At line 9 we collect a new starting vertex for the new cluster. Lines 10..13 are almost the same as lines 22..25: they add the current vertex to the worklist and put it in a cluster. In the `while`-loop on line 14 we perform the depth-first search. In lines 18..30 each vertex is handled.

If the vertex is not yet in a cluster, it is added to a cluster and to the worklist. If it is already in a cluster and even the cluster we are now generating, there could be a cycle. This possibility is checked by `handle_cycle()`, discussed in section 5.6.

Building cluster packages is done as follows. Each vertex has a list of outgoing edges. An outgoing edge is described by the destination vertex ID. Before the cluster packaging, this is the global vertex ID, but to make it a useful package, we need to convert this to its local vertex ID. This can be done using the `vertexInCluster` vector we built earlier. If a vertex has an edge to another vertex outside the cluster, we call the second vertex a foreign vertex. The associated edge is a foreign edge. Foreign outgoing edges are not converted to the local vertex ID, but are converted to the negative of the global vertex ID. And so, an SPU knows when an edge is local or foreign. Thus, it knows when to fetch data from its local store or from the main memory.

```
1  foreach(cluster CL) {
2    for(vertex_id vid = 0; vid<CL.size; ++vid) {
3      glo_vid = CL.vertexIDs[vid];
4      CL.vertices[vid] = graph[glo_vid];
5      foreach(vertex_id S:
6              CL.vertices[vid].outEdges) {
7        CL.vertices[vid]. out =
8          vertexInCluster[S]==cluster_id ?
9            vertexIDInCluster[S] :
10           -S ;
11     }
12   }
13 }
```

5.2 Lifting order

For this research, only one heuristic describing the lifting order of vertices was used: the *swiping* heuristic. The following pseudocode describes this heuristic.

```
1  int next_vertex = 0;
2  bool vertices_lifted = false;
3  unsigned int get_next_vertex() {
4    int vertex = next_vertex++;
5    if(vertex>=cluster.size) {
6      if(vertices_lifted) {
7        vertex = 0;
8        next_vertex = 1;
9        vertices_lifted = false;
10     }
11   }
12   return vertex;
13 }
```

The `vertices_lifted` boolean is set to `true` in the next pseudocode, to signal there was a vertex lifted in the last iteration. When that happens, the swiping starts over as per lines 7..9.

The general lifting algorithm is as follows.

```
1  read_from_main_mem(cluster_measures);
2  while((vid = get_next_vertex()) < cluster.size) {
3    vertex V = cluster.vertices[vid];
4    if(cluster_measures[vid]==Top) continue;
5    measure_t M = get_measure(V.outEdge[0]);
6    for(int i=1; i<V.outEdges; ++i) {
7          measure_t T = get_measure(V.outEdge[i]);
8      if(V.player==ODD) {
9        if(measure_cmp(T,M)>0)
10         M = T;
11     } else {
12       if(measure_cmp(T,M)<0)
13         M = T;
14     }
15   }
16   measure_t R = succ(M);
17   if(measure_cmp(R,cluster_measures[vid])>0) {
18     cluster_measures[vid] = R;
19     vertices_lifted = true;
20   }
21 }
22 write_to_main_mem(cluster_measures);
```

In line 1 we first copy the measures from the main memory to the local storage of the SPU. In lines 5..16 the actual lifting is performed, going through all the edges and calculating the successor of the found edge. In the mathematical algorithm, the successor of every edge measure is calculated before the comparison in lines 9 and 12. However, calculating the successor can also be done after the comparisons, bringing down the number of times a successor is calculated. This increases performance. In lines 17..20 the resulting measure is compared with the current measure. If needed, the measure is updated and it is remembered a lift was successfully performed. This is done by setting `vertices_lifted` to `true`, which is used in the `get_next_vertex` function. This function is described in the previous pseudocode. Finally all the resulting measures are written back to the main memory in line 22.

This code relies on the following function. It uses the information we put in the signedness of the vertex ID earlier, to determine if the vertex is local of foreign.

```
1  measure_t get_measure(int vertex_id) {
2    if(vertex_id>=0) {
3      return cluster_measures[vertex_id];
4    } else {
5      return get_foreign_measure(-vertex_id);
6    }
7  }
```

5.3 Cluster order

For this research, only one heuristic describing the issuing order of clusters was used: the *swiping* heuristic. Its implementation is very similar to the implementation of the swiping heuristic for the lifting order of vertices.

5.4 Capturing cycles

The following pseudocode describes the regeneration of clusters. After 20 iterations, the clusters are regenerated, alternating between two methods of searching for a new starting vertex. Only clusters that have been lifted in the last iteration are regenerated. This avoids regenerating parts of the graph that are not or hardly active. The number of clusters regenerated can vary, but usually the number gets smaller in time. This is because the algorithm is closer to the solution, so nearer the fix point. This means less vertices have to be lifted and thus, less clusters in general.

```
1  vector clstr_lifts;
2  while(!algorithm_finished) {
3    for(int runs=20; runs--;) {
4      for(cluster CL) {
5        clstr_lifts[CL] = lift_cluster(CL);
6      }
7    }
8    if(technique==FRONTTOBACK) {
9      generate_clusters(BACKTOFRONT, clstr_lifts);
10   } else {
11     generate_clusters(FRONTTOBACK, clstr_lifts);
12   }
13 }
```

5.5 Lifting vertices

The following C code is an implementation to check if two measures are equal using the altivec instruction set. Line 4 contains the altivec instruction used [5]. Note that this function only checks for equality.

```
1  typedef vector unsigned int measure_t;
2  int measure_cmpeq( const measure_t& a,
3                     const measure_t& b) {
4    measure_t d = spu_cmpeq(a,b);
5    return ~( ((unsigned int*)&d)[0] &
6              ((unsigned int*)&d)[1] &
7              ((unsigned int*)&d)[2] &
8              ((unsigned int*)&d)[3] );
9  }
```

A more general function also returning whether measure `a` is greater than or lesser than measure `b` could be as follows. Altivec instructions are on lines 3 and 4. Since components with a lower index are more important, the comparison starts at index 0 and goes up (lines 5..10). As soon as a difference is found, the result can be returned.

```
1  int measure_cmp( const measure_t& a,
2                   const measure_t& b) {
3    measure_t dab = spu_cmpgt(a,b);
4    measure_t dba = spu_cmpgt(b,a);
5    for(int i=0; i<4; ++i) {
6      if(((unsigned int*)&dab)[i])
7        return 1;
8      else if(((unsigned int*)&dba)[i])
9        return -1;
10   }
11   return 0;
12 }
```

By comparison, the second function could also be implemented without using the altivec instruction set. The implementation of the corresponding function is shown below. In this implementation, it is easier to understand how it works.

```
1  int measure_cmp( const measure_t& a,
2                   const measure_t& b) {
3    for(int i=0; i<4; ++i) {
4      if( ((unsigned int*)&a)[i] >
5          ((unsigned int*)&b)[i] )
6        return 1;
7      else if( ((unsigned int*)&a)[i] <
8               ((unsigned int*)&b)[i] )
9        return -1;
10   }
11   return 0;
12 }
```

5.6 Graph Preprocessing

The first part of the preprocessing was borrowed from Weber and Van de Pol [12] and Verver [13].

```
1  for(vertex V: vertices) {
2    bool remove_self_edge = false;
3    bool remove_other_edges = false;
4    for(edge S: V.outEdges) {
5      if(S==V) {
6        if( (V.priority&1) &&
7            ( (V.player==ODD) ||
8              (V.outEdges==1)) ) {
9          V.measure = Top;
10       }
11       if((V.priority&1) == V.player) {
12         remove_other_edges = true;
13       } else if(V.outEdges > 1) {
14         remove_self_edge = true;
15         continue;
16       }
17     }
18   }
19   for(edge S: V.outEdges) {
20     if( (S == V && remove_self_edge) ||
21         (S != V && remove_other_edges) ) {
22       V.outEdges.del(S);
23   }
24 }
```

The preprocessing assigns the $\top$ measure to vertices that are certain to become $\top$ after the algorithm (lines 6..10). It also determines from which vertices to remove unnecessary edges in lines 11..16. The actual deletion is done in lines 19..23.

The second part of the preprocessing is performed during the clustering discussed in section 5.1. It detects cycles within clusters and preprocesses them if a solution is known beforehand. The pseudocode is show below. The outer `for`-loop starts at the back of the worklist stack, i.e. the deepest vertex on the stack. This `for`-loop stops in two cases. In the lines 3..6 we check if the cycle is of the kind we want. If it is not, we break out of the loop. In lines 7..12 for the actual cycle. If it is found, we assign the $\top$ measure to each vertex in the cycle.

```
1  handle_cycle(C, cluster_id, worklist) {
2    for(int vid=worklist.size(); vid--; ) {
3      if( (worklist[vid].player!=PLAYER_ODD) ||
4          !(worklist[vid].priority&1)) {
5        break;
6      }
7      if(worklist[vid]==C) {
8        for( int vid2=vid; vid2<worklist.size();
9             ++vid2) {
10         vertices[vid2].measure = game->top_vector;
11       }
12     }
13   }
14 }
```

6. BENCHMARKS

The following benchmarks were made using k-bounded clustering, k=8 and clusters of 400 vertices. The vertices in all the used graphs have 4 outgoing edges. Note that the number of vertices doesn't say much about the complexity of the graph. They are only listed to distinguish the graphs. The real complexity is determined by the relative position of the vertices. Cycles add much to the complexity. The graphs were randomly generated, using the graph generation procedure described by Kandziora [9].

6.1 General performance

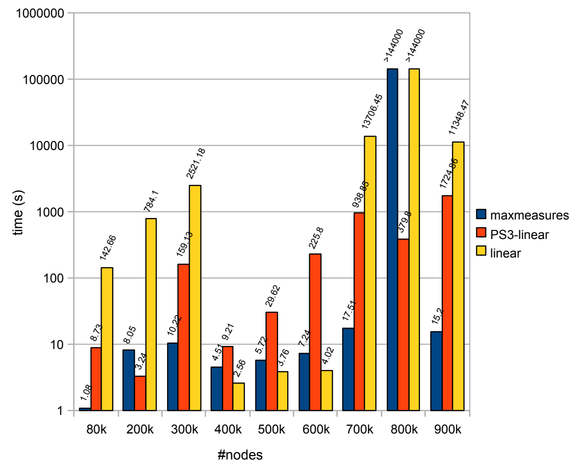

Figure 3. Solving time of various implementations and graphs of various sizes

When we compare the total time it took to solve a graph with the total time it took Verver's program to solve the same graph, we get a general impression of the performance. Two lifting strategies of his program were used for the comparison: a *linear* strategy, similar to *swiping*, and the so-called *MaxMeasures* strategy. Figure 3 shows the results.

Verver's program was run on an AMD Sempron 2400+ CPU on an Asus A7V8X-X motherboard with 1GB of PC2700 RAM. The operating system was GNU/Linux, using Linux 2.6.33.

The results show that the MaxMeasures of Verver is significantly faster than the current implementation on the PS3. It is a lot faster, even though it's running on a singlecore 1.7GHz CPU and the PS3 contains seven 3GHz processors. Of course, the PS3 cores are distributed. Moreover, the PS3 implementation uses only a linear heuristic. When we compare to Verver's implementation using a linear heuristic, we see that the PS3 has real potential.

Especially noteworthy is the results of the game of 800k vertices. Verver's program, using either the MaxMeasures or linear heuristic, was not done solving after 40 hours. At this point the benchmark was halted. The PS3 implementation took 6 minutes and 20 seconds to solve the same game. This is a substantial difference. It can be explained by the detection of cycles. The PS3 implementation detects cycles that would take a lot of lifting, whereas Verver's program does not. When running the benchmark without the cycle detection, the PS3 implementation does not finish in 10 hours either. Even though the MaxMeasures algorithm focuses on lifting these cycles, they take a lot of time because the $\top$ measure of this game is $\{113791, 112338, 113028, 113088\}$. The number of lift operations required to lift each of these cycles, is about the product of the components of the $\top$ measure.

A more usual trend is for example the game of 700k vertices. Here we see that the MaxMeasures heuristic is the fastest by far and the PS3 linear implementation the second fastest. The slowest is the linear implementation of Verver, taking more than 14 times as much time as the PS3 linear implementation. The same trend can be seen in other benchmarks, e.g. the games of 80k vertices, 300k vertices and 900k vertices.

6.2 Graph Preprocessing

The benchmarks in Table 2 and Figure 4 show the influence of preprocessing a graph. It is clear that preprocessing is vital to a fast execution of the algorithm. Even though it is only one benchmark, the difference is huge. Preprocessing reduced the solving time of some graphs from 12 hours to 10 seconds. This particular parity game is solved over a thousand times faster. The reason is the self-loop. The algorithm needs a lot of time lifting the vertex with the self-loop, even though beforehand it would be known the result would be the $\top$ measure. The preprocessing catches this and assigns the $\top$ measure to the vertex, saving a lot of time. This particular graph had a $\top$ measure of $\{1073, 1140, 1115, 786\}$, so without preprocessing it would take about $1073 \cdot 1140 \cdot 1115 \cdot 786 \approx 10^{12}$ lift operations for this one vertex to be lifted.

Table 2. Preprocessing on a graph containing a vertex with self-loop

No preprocessing	With preprocessing
515,44s	0,14s

6.3 Number of SPUs

The benchmarks in Figure 5 show the influence of the number of SPUs at work. The benchmarks include only the solving time, not the preprocessing or the clustering time. They are not included because the number of SPUs only affect the solving time.

Our intuition was right: the more SPUs we use, the faster the algorithm is performed. An interesting result is the time it took to solve the game of 200,000 vertices using one SPU. One would expect a solving time of roughly 15 seconds, extrapolating from other results, but instead it took 137 seconds to solve this game. A possible explanation might be that the cluster ordering is unlucky, causing adjacent clusters to be lifted far apart. This would account for more iterations. However, both solving instances take 10 iterations, so until further investigation it will remain an anomaly.

The game of 400,000 vertices is a more general illustration of the influence of the number of SPUs. Here we clearly see that doubling the number of SPUs almost cuts the solving time in half. This means the EIB is not taxed to its maximum capacity and the bottleneck is the algorithm itself. When more optimizations would be implemented, this could reverse. The SPUs would demand more of the EIB and we would see a more shallow curve near the end, because more SPUs means more simultaneous memory transfers.

This happens with the game of 500,000 vertices. Between one and two cores the time is halved, but between one and three cores the time is not divided by three. Moreover, time is not divided by six when using six cores. This could mean a taxed EIB or an unlucky circumstance, like with the game of 200,000 vertices.

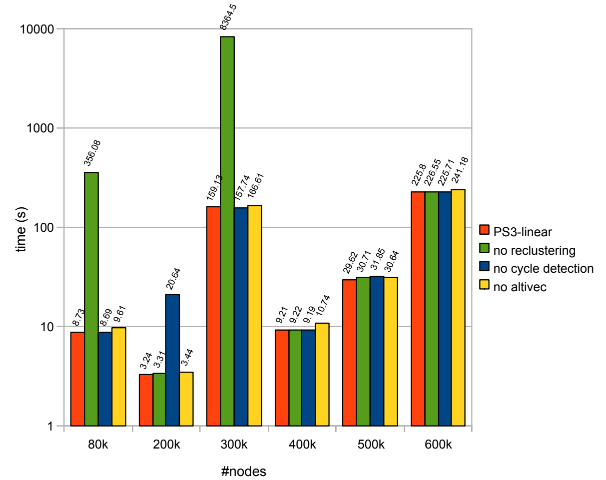

Figure 4. Comparison of solving times of various optimizations

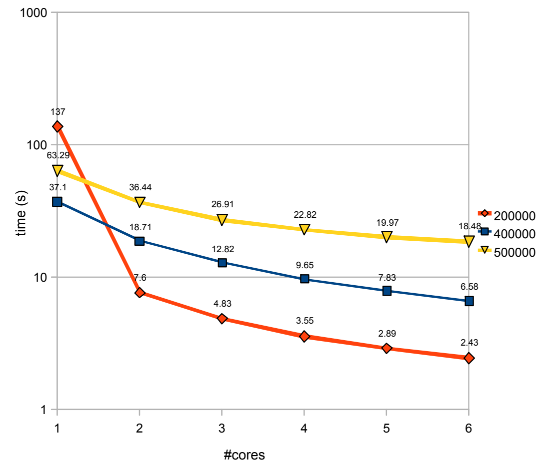

Figure 5. Influence of the number of cores used

6.4 Reclustering

The benchmarks in Figure 4 show the reclustering can be beneficial when a cycle is not caught on the first clustering. The solving times of the games with 80,000 vertices and 300,000 vertices are considerably reduced using this technique.

6.5 Altivec instruction set

The influence of using the altivec instruction set is shown in Figure 4. The improvement is between 5% and 6% for the selected parity games.

7. CONCLUSIONS

We have determined that optimizing the Small Progress Measures algorithm for the Playstation 3 has six important aspects: graph subdivision, lifting order, cluster issuing order, capturing cycles, the lifting heuristic and preprocessing the graph. These six aspects are important because they affect the performance of the algorithm significantly.

Various optimizations are proposed, e.g.: cluster regeneration, altivec instructions set implementation, and the cluster dependency. Some of these were implemented and benchmarked. The benchmarks show in its current implementation the algorithm is not up to par with other implementations, like MaxMeasures. However, much optimization is still possible. When comparing with x86 using similar heuristics, the implementation is significantly faster. Because of the distributive nature of the Cell CPU, this algorithm could be applied to a cluster of Playstation 3 units. Therefore, the Playstation 3 seems a viable architecture for the Small Progress Measures algorithm, but further research is needed.

8. FUTURE

Not all the optimizations proposed were implemented, but would expectedly improve performance. For example the focus list for the clusters issuing order is worth further investigation. Looking into incorporating MaxMeasures for the vertex lifting order is also interesting, seen as this heuristic is a significant improvement over the linear heuristic. When assuming all cycles can be individually fitted on one SPU, it is interesting to look at detecting cycles beforehand and clustering the graph accordingly.

When reusing the current implementation, the first priority should be to fix the limitation of 512 vertices per SPU. Secondly the current implementation requires too many DMA requests to the main memory. These should be buffered more. Then further optimizations can be investigated. There is a preliminary implementation of a focus list, so this can be utilized.

9. ACKNOWLEDGMENTS

I'd like to thank Michael Weber for his advice and guidance, aiding me in the research. He also provided a nice example of a parity game, which I borrowed.

Secondly I'd like to thanks Maks Verver, without his implementation on the x86 I would not have a way of verifying correctness nor a basis for comparison.

10. REFERENCES

[1] D. Bošnački, S. Edelkamp, and D. Sulewski. Efficient probabilistic model checking on general purpose graphics processors. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*, pages 32–49, Berlin, Heidelberg, 2009. Springer-Verlag.

[2] T. Halfhill. Microprocessor report: Ambric's new parallel processor. oct 2006.

[3] D. Harel and R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[4] IBM. Cellsdk, `http://www.ibm.com/developerworks/power/cell/downloads.html` [last checked: March, 2010].

[5] IBM. Synergistic processor unit instruction set architecture, `https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/76CA6C7304210F3987257060006F2C44/\$file/SPU_ISA_v1.2_27Jan2007_pub.pdf` [last checked: May, 2010].

[6] IBM. Cell broadband engine architecture. technical report version 1.02. technical report, ibm systems and technology group. oct 2007.

[7] Intel. Intel teraflops reasearch chips, `http://techresearch.intel.com/articles/Tera-Scale/1449.htm` [last checked: March, 2010].

[8] M. Jurdzinski. Small progress measures for solving parity games. In *In 17th Annual Symposium on Theoretical Aspects of Computer Science*, pages 290–301. Springer, 2000.

[9] J. Kandziora. Playing parity games on the playstation 3. Master's thesis, University of Twente, jan 2009.

[10] nVidia. Geforce gtx 295, `http://www.nvidia.com/object/product_geforce_gtx_295_us.html` [last checked: March, 2010].

[11] nVidia. nvidia's next generation cuda compute architecture: Fermi v1.1. 2009.

[12] J. van de Pol and M. Weber. A multi-core solver for parity games. *Electron. Notes Theor. Comput. Sci.*, 220(2):19–34, 2008.

[13] M. Verver, M. Weber, and J. van de Pol. Maxmeasure.

[14] T. Wilke. Alternating tree automata, parity games, and modal mu-calculus. *Bull. Belg. Math. Soc.*, 8(2):359–391, 2002.