Schedae Informaticae Vol. 24 (2015): 179–195
doi: 10.4467/20838476SI.16.016.4357

Java Based Transistor Level CPU Simulation Speedup Techniques

Tomasz Wojtowicz
Department of Computer Sciences and Computer Methods, Pedagogical University
ul. Podchorążych 2, 30-084 Kraków, Poland
e-mail: *tomaswoj@gmail.com*

Abstract. Transistor level simulation of the CPU, while very accurate, brings also the performance challenge. MOS6502 CPU simulation algorithm is analysed with several optimisation techniques proposed. Application of these techniques improved the transistor level simulation speed by a factor of 3–4, bringing it to the levels on par with fastest RTL-level simulations so far.

Keywords: CPU, microarchitecture, simulation, registers, pipeline, activity, 6502.

1. Introduction

With a software industry reaching a certain maturity level and the growing amount of legacy software (and computer system platforms needed to execute it) there is a movement recently for software preservation [1, 2]. These efforts often involve creation of execution platforms for the old software that could emulate or even simulate with high fidelity the hardware that is no longer available [3]. While Instruction Set Architecture (ISA) level emulation provides high emulation speeds (often able to execute the code much faster than the original platform) it still lacks the high fidelity aspect. Very often, for the software that exploited undocumented platform capabilities or that relied on specific timing characteristics, a special handling in the emulation code needs to be provided to make it run. That result in long development time and high effort required to create a reliable emulation at ISA level. Therefore just recently there are research activities emerging in leveraging low level simulation as the emulation platform [4, 5]. As long as there is a way to reverse engineer the

actual physical design of the chip, either through access to its original design documentation or through decapping of the actual physical chip - it shall be possible to turn this information into a working simulator of that chip, capable of running the software originally written for the platform. On the more forward looking side there is a brand new world of the Internet Of Things (IoT) emerging, with microchips being embedded into everything, wearable computing, small IP stacks, microcontrollers in everything, etc. Many of the solutions here involve dedicated chips, or a bespoke product combined of multiple small Commercial Off The Shelf (COTS) chips and a dedicated set of software written at the very low level due to the resource constraints on these platforms [6, 7]. An effective emulation platform is needed here to enable early software creation in parallel with hardware development, to bring down the time to market as much as possible. In both areas there is an opportunity to leverage low level simulation, instead of ISA level emulation, or some mixed approach that involves low level simulation for more complex or custom subsystems, while ISA or other relatively high level emulation is used for generic subsystems and components. The challenge however with low level simulation is usually on the speed side. Transistor level simulators tend to be slow and of limited usability if near real-time software execution is required. A faster alternative can be Register Transfer Level (RTL) simulation, but that on the other hand requires an integrated circuit design documentation to be available, and that is often not the case for legacy platforms. In this work a single-threaded, transistor level simulator of a classic 8-bit CPU is presented, implemented in Java, originally running at a speed of approx. 0.9 kHz on a desktop PC. The original algorithm is then refined and initial Java implementation refactored, taking performance into account what eventually enables the simulator to run at the speed of 3.2–3.5 kHz. This improved speed is comparable with the 4 kHz speed of the RTL simulator of the same processor [8].

The paper is organised as follows: first previous works and research in this area are presented, with a focus on the original works on reverse engineering and simulation of the MOS6502 CPU; then the original model and transistor level simulation algorithm is presented and its execution is analysed. In the next section some improvements are proposed both at the algorithmic level and at the Java programming language implementation level with focus on the simulation speed. In the last section improvement results are presented and future work is discussed. Note that while the paper focuses on the specific CPU, the simulator itself can be easily extended to cover other CPUs, like Motorola 6500, Zilog Z80, Intel 8086 or dedicated chipsets found in the legacy microcomputer platforms.

1.1. Previous works

In the recent years at least several research efforts have started to use high fidelity simulation of integrated circuits or microprocessors for emulation purposes. There is a DICE project (Discrete Integrated Circuits Emulator) [4] that targets old 1960s and 70s game consoles. The simulation there is implemented at the level of individual Transistor-Transistor Logic (TTL) components. On a modern PC with a 3GHz CPU

this simulator is capable of running a relatively simple Pong or Breakout game console at near real-time. On the other hand there are Field Programmable Gate Array (FPGA) based prototypes that try to simulate the circuits via a properly programmed FPGA board. These projects like Amiga Minimig [9] or Atari ColdFire [10] are capable of running the emulated platform even faster than the originals, keeping all the nuances of the original hardware. The down side here is that they require dedicated hardware, FPGA boards like Xilinx Spartan or Altera Cyclone. One of the most impressive efforts recently was reverse engineering of the MOS6502 CPU, via decapping and digitalization of high resolution photographs of the different layers of this CPU [5]. This resulted in an accurate CPU model created at the transistor level. Part of that project was also to develop a working simulator of that CPU, capable of executing an original binary code with a reasonable (but not even close to real-time) speed. The original simulator implementation in JavaScript was capable of running the CPU at 1Hz level in a web browser. Further advancements in browser technologies allowed the model to run at approx. 250Hz speed (assuming all the visualisation was disabled). Several ports of the simulator followed, with low-level C implementation capable of reaching around 1-1.5 kHz. The fastest reported software implementation involved a translation of the transistor level netlist into an RTL level design of the CPU, and running such a model in a Verilator. Moving the level of abstraction up allowed that RTL model to reach the speed of approx 4 kHz on the modern PC [8].

1.2. Proposed approach

In this paper authors Java based implementation of the 6502 simulator described above is taken as a starting point [11]. This implementation runs on a modern desktop PC (Intel i5, 2.6GHz, 8GB RAM, Windows7 64bit) at a speed of approx 0.9 kHz (on JRE 1.7, 64bit). Runtime characteristics of the simulator are gathered and analysed. The improvement recommendations are given in two main aspects:

- Java language and platform: as Java is a managed runtime environment, that involves for example translation and compilation (JIT) of the byte code into the native code in runtime, garbage collection, use of specific platform libraries (e.g. Java collections) – there are some recommendations given at the programming language level that improve the performance of the simulator [10]. These recommendations can be helpful for any other low level simulator written in Java, they are by no means this simulator or the particular CPU model specific.
- Simulation algorithm: these are the recommendations and improvements based on the analysis of this particular simulation algorithm applied to the particular CPU (6502). It means that while they may be beneficial to other CPU models, it shall be carefully measured for other CPUs. Most likely comparable CPUs (like Motorola 6500 or Intel 8086) as they share in general a similar architecture will exhibit a similar behaviour and thus will react to the same set of improvements.

To ensure the improvements are systemic and not just for corner cases, several factors were taken into account:

- Java JVMs, due to JIT are known for so called ramp-up times. That means that first phases of code execution can be a bit slower, but once the JIT compiler settles, and has a relatively long trace of code execution, it will optimise it resulting in a higher execution speeds later on. The ramp-up time varies between applications. It depends on the application type - it can be much longer for web server based applications (even in range of tens of minutes), but it also depends on whether a particular part of the application code was executed already or not. In case of the CPU simulator the ramp-up time is relatively low, and is in range of 10–15s at most. Also, by the nature of the simulator, vast majority of its code is executed straight from the first cycles of the simulated CPU. Therefore while the reliable measurements of the speed need not to be taken on early cycles; it is expect the speed will stabilise after several thousands of cycles.
- Related to the above is the impact of Garbage Collector (GC) on the simulator speed. In implementations leveraging a dynamic creation of objects in runtime that may become a serious factor and may result in periodic speed drops during the simulation, when the GC 'kicks in'. That at some point later could result in speed 'hiccups' visible to the user, especially if the simulated speed reaches the interactive, real-time levels.
- To ensure that model analysis and improvements are not exploiting characteristics of certain binary code that is run through the simulator - at least several 6502 benchmark programs are used during the speed measurements.
- To ensure that improved implementation is not introducing errors to the simulation core – it is tested against the initial, reference core implementation. For at least several thousands of cycles all the key registers, address buses and data buses are checked with the reference implementation.

2. Original algorithm and its analysis

MOS6502 is an 8-bit little endian general purpose processor with an 8-bit data bus and 16-bit address bus. Therefore it can directly access 65535 bytes of RAM memory, but higher amounts of memory are also supported via the bank switching. It was originally running with speeds range of 1–2MHz. It was usually delivered in a 40-pin DIP package. At the integrated circuit level it consists of approx. 3500 depletion-mode MOSFET transistors, resulting in approximately 1400 logic gates.

2.1. Simulator overview

Decapping the 6502 and taking high resolution transistor level imagery of CPU layers gave the community an access to the complete CPU netlist [5] that can be loaded to the simulator software and enable the simulation at the single transistor level.

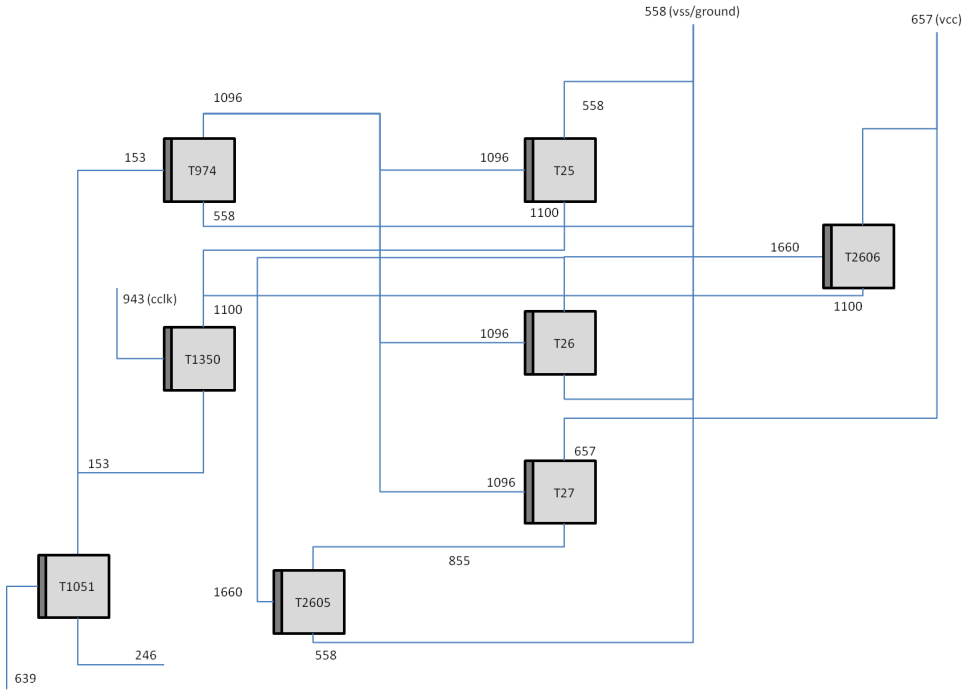

Figure 1. Excerpt of CPU schematics with transistor netlist.

The diagram above (Figure 1) shows a small excerpt of CPU schematics with transistors interconnected by a set of segments (dark grey parts of transistors denote gates, other connections are C1 and C2). The corresponding netlist is specified in 2 datasets of form (excerpt):

```
[ 364,'-',1,1593,2562,1525,2562,1525,2591,1593,2591],
[ 365,'+',1,8232,6988,8152,6988,8152,7011,8198,7011,8198],
[ 365,'+',1,8461,7038,8484,7038,8484,6990,8440,6990,8440],
```

for segments (connections between transistors, and special segments like VCC, VSS or CLK). The first parameter is for segment ID, the second stands for default segment state, the third is for layer where the segment is in the layout (relevant for visualisation only), followed by the polyline defining the physical layout of the segment on the board. Multiple definitions for the same segment indicate more complex physical implementation (multiple polygons on the board).

The second dataset of form (excerpt):

```
['t24', 710, 1495, 348, [7373, 7394, 5351, 5380]],
['t25', 1096, 1100, 558, [1373, 1434, 3928, 4213]],
['t26', 1096, 558, 1660, [1267, 1285, 4048, 4139]],
['t27', 1096, 855, 657, [1197, 1241, 3940, 3957]],
['t28', 1503, 558, 744, [6928, 6974, 4477, 4570]],
```

is for the actual transistors. The first value in the line denote transistor ID (name), second is it's gate segment, third and fourth are its corresponding C1 and C2 segments. The lists that follow are transistor bounding boxes (for visualisation purposes only). Both datasets are loaded into the simulator at start up and dynamically wired in software. That means any alternative CPU could be loaded and simulated if only the complete netlist is available.

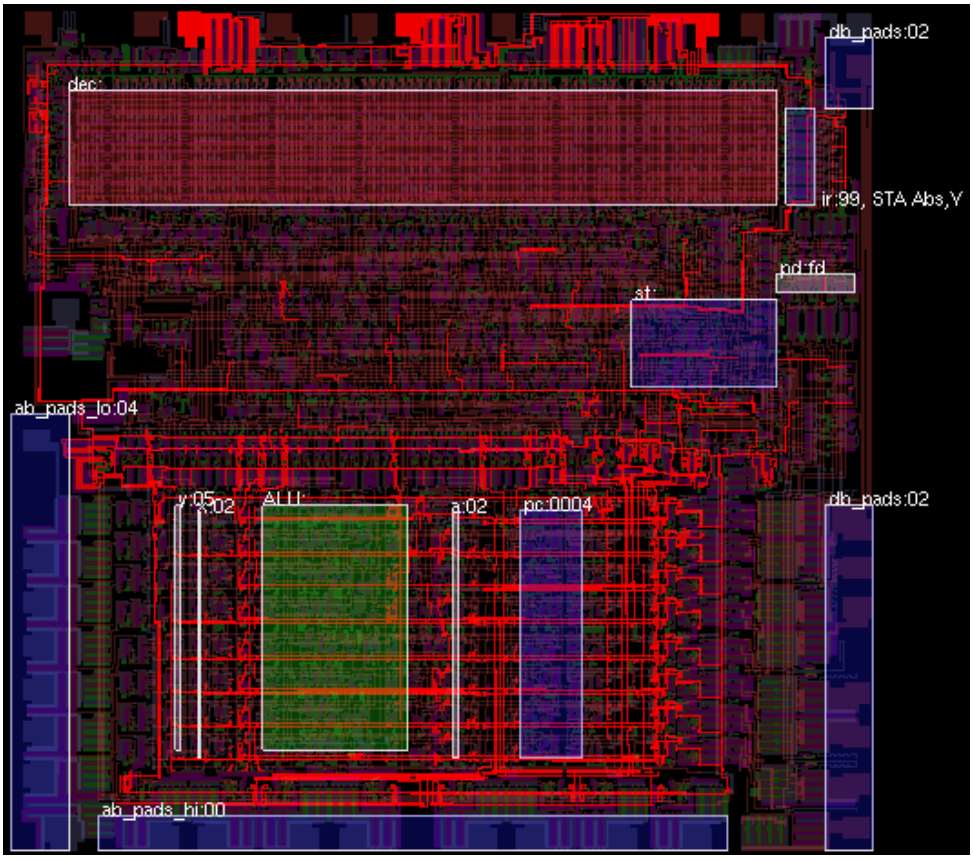

Figure 2. Visualization of CPU internals.

As part of [11], based on the original JavaScript code [12], a Java based implementation of this CPU model was built. Original algorithm was implemented using Java collections, with the focus being on the visualisation of the CPU internals (see Figure 2). Focus of the current paper is on the simulation performance.

2.2. Simulation algorithm overview

The best way to understand the simulation algorithm to be optimized is to present it in a top-down approach. MOS6502, as most of traditional CPUs, is a sequential, synchronous integrated circuit being driven by the clock (CLK) and the current state of its inputs (with data bus and address bus being the most important from the functional perspective). Therefore the top level loop pseudocode looks as follows:

```
halfStep()
{
 if (CLK==HIGH)
  {
   CLK=LOW;
   recalcSegmentList(CLK);
   if (CPUstate.allowsDataBusWrite)
   {
    writeDataBus(memory[readAddressBus()]);
    recalcSegmentList(DATA_BUS);
}
  }
  else
  {
   CLK=HIGH;
   recalcSegmentList(CLK);
   if (!CPUstate.allowsDataBusWrite)
   {
 writeMemory(readDataBus(), readAddressBus());
   }
  }
}
```

Key next level procedure is recalculating the state of the CPU based on modified inputs (like CLK – clock or the data bus – that is populated with data fetched into the CPU pads from a proper place in memory). Its high level pseudocode is shown below:

```
recalcSegmentList(listOfSegments)
{
 nextIterationList = new List();
 while (listOfSegments.size>0)
 {
  foreach (segment in listOfSegments)
  {
   recalcSegment(segment);
   //that involves populating nextIterationList
  }
  listOfSegments = nextIterationList;
```

```
  nextIterationList = new List();
 }
}
```

The above pseudocode highlights the iterative nature of the algorithm. Usually it takes more than a couple of iterations each half cycle (on each switch of the CLK value) to stabilise the whole CPU state – and then *listOfSegments* becomes empty. From the measurements the number of iterations required to reach the stable CPU state (stable state of all its segments and transistors) is between 10 and 18 and depends on the size of the CPU itself (the amount of segments and transistors). Within each of the iterations a list of segments possibly switching the state is evaluated. This is performed by the following function:

```
recalcSegment(segment)
{
 //there is no point to evaluate
 //ground (VSS) or power (VCC)
 if (segment == GND or PWR) return;

 segmentGroup = new List();
 addSegmentToGroup(segment);
 newState = getGroupValue(segmentGroup);
 foreach (segment in segmentGroup)
 {
  if (segment.state!=newState)
  {
   Segment.state = newState;
   foreach (tr where tr.gate=segment)
   {
    if (tr changes to on)
    {
     addToNextIterationList(tr.c1);
    }
    if (tr changes to off)
    {
     addToNextIterationList(tr.c1);
     addToNextIterationList(tr.c2);
    }
   }
  }
 }
}
```

The above pseudocode first creates a group of segments (*segmentGroup*) interconnected via enabled transistors with the examined segment. Then based on the members of that group calculates the new value of the group. When the value is established, all segments in the group that have a different value are checked whether they are gates for any transistor (that in turn would mean the transistor state needs

to change). Segments that are impacted by transistor state changes are added to the recalculation list for the next iteration. When the recalculation list is empty – the CPU reaches a stable state. The segment group creation is implemented as follows:

```
addSegmentToGroup(segment)
{
 if (segmentGroup.has(segment)) return;
 if (segment==VSS or VCC)
 {
  segmentGroup.add(segment);
  return;
 }
 foreach (transistor in segment.c1c2List)
 {
  if (transistor.state=on)
  {
   if (tr.c1=segment) addSegmentToGroup(tr.c2);
   //note the recursion!
   if (tr.c2=segment) addSegmentToGroup(tr.c1);
  }
 }
}
```

Note that in the original algorithm this is a recursive function (as highlighted above). Finally a pseudocode to evaluate value of the group:

```
getGroupValue(segmentGroup)
{
 if (group.contains(GND)) return false;
 if (group.contains(PWR)) return true;
 if (group.contains(segment.state=pullup))
   return true;
 return false;
}
```

The algorithm every CLK cycle iterates through the integrated circuit, taking the changes on the CPU pads and evaluating the new state of the segments and transistors. To some extent it resembles event based simulation methods that works on the logic gate level, but in this case the 'events' are happening at transistor and segment level.

2.3. Internal algorithm statistics and observations

Profiling of the original implementation code exposed the following CPU usage.

Large part of the simulation is spent in *addSegmentToGroup()* method. Another significant contributor is a method that encapsulates the addSegmentToGroup – *recalcSegment()*. That means these methods are the hotspots of the simulation, and optimising these shall give some speed ups.

Table 1. CPU time spent in various methods of the simulation (JProfiler, sampling profiling method)

Method	CPU[%]
halfStep()	99.6
-recalSegmentList()(CLK to low)	−44.2
–recalcSegment()	−41.2
—addSegmentToGroup()	−26.2
-recalcSegmentList()(CLK to high)	−37.4
–recalcSegment()	−34.8
—addSegmentToGroup()	−22.6

Looking at *addSegmentToGroup()* it is worth to analyse how the recursion scheme is used. It turn out that the size of the groups created is extremely small (comparing to the size of the whole chip), and tends to be shifted into a low single digit values.

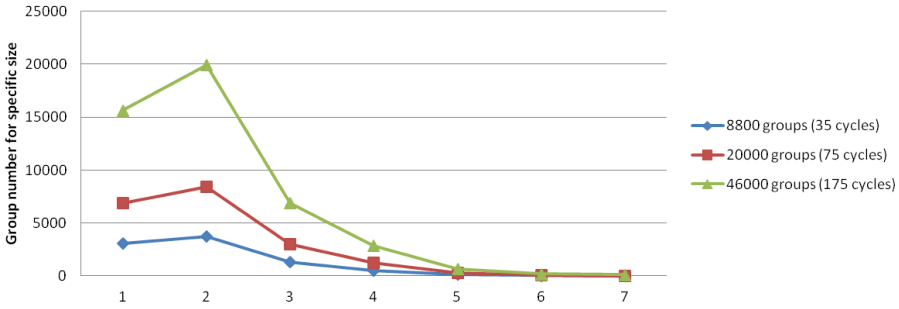

Figure 3. Group size distribution.

The chart (Figure 3) shows the amount of groups of certain size (from 1 to 7 segments, no groups of 8+ segments were spotted), as captured over time, in 3 different series (where 46000 groups series corresponds to approximately 80–90 full CPU cycles). This highlights the fact that majority of groups created (vs. all groups) is of 1–3 size. That means there is a relatively high overhead in recursive call of *addSegmentToGroup()* method comparing to the size of the group. Another observation is that there are a relatively high number of groups created per cycle (on average around 260 groups). That means there is a massive amount of calls to that method per cycle, multiplied by recursive calls. As each recursive call involves performing some additional operations on the stack – flattening this function, and replacing the recursive calls with iterative approach should yield some performance benefits.

Another observation is that the actual implementation of many of the highlighted

methods either relies on Java collections (like ArrayList or Hashmap), that are dynamically created on each iteration, or assuming the basic static (in size) lookup arrays are used – requires proper cleaning of these lookups between the iterations:

- In the creation case, while the code looks clean there is obviously an overhead of collection implementation and creation of the new collection objects. Moreover occasionally GC kicks in, trying to retain the memory from objects that are no longer referenced and used.
- Even if the Java collections are used only for static collections (e.g. global list of segments or transistors – that are referenced by name or by identifier from inside the algorithm) – fetching them using Collection.get(key) is an additional overhead to the simulator.
- In the second case, when flay arrays are used instead of collections, all the performance gains can be lost, if these arrays needs to be cleaned up every iteration (especially inside the *addSegmentToGroup()* method) in a brute force way (or re-allocated). Algorithm analysis shows there are at least several places, where some information can be reused between iterations (memoization), to avoid the need of whole array reinitialisation or creation of new object.

Another observation is that more than 50% of all the groups created as part of *recalcNode()* contains the VCC segment (power) or VSS segment (ground). That observation can be leveraged in at least several places, structuring the checks vs. these segments properly, and pushing them up in the method body whenever possible. That allows cutting method code execution as early as possible. Based on these observations some improvements are proposed that as a result increased the simulation speed by a factor of 3–4.

In practise what works best is a combination of functional bounding boxes and data paths between these functional areas. Therefore in our visualisation framework this is all configurable by the user.

3. Proposed improvements

3.1. Optimisations specific for Java language and runtime environment

3.1.1. Refactoring of addSegmentToGroup() method

As noted above, this particular method takes almost 50% of the whole simulation CPU time. Therefore this method is considered as a hotspot and refactored to use iteration instead of recursion. Moreover group list implemented with ArrayList is replaced with standard integer array – *groupArray*. Also for each new group the old

groupArray is reused, without any cleanup, thanks to use of proper indexes. The refactored method will become of form:

```
addSegmentToGroup(segment)
{
 //leverage previous group information
 //to clean the groupContains lookup
 for (int i=0;i<groupArrayMax;i++) groupContains[i]=false;

 // proper play with indexes allows to reuse groupArray
 // without a need to clean it for every new group
 groupArrayMax=0;
 addedPos=0;
 tabletoAdd[0]=segment;
 maxPos=1;

 //flags for ahead of time group value calculation
 groupTrue=false;
 groupFalse=false;

 //this allows to eliminate recursive calls
 while (addedPos<maxPos)
 {
  segmentId=tableToAdd[addedPos];
  if (groupContains[segmentId]
  {
   addedPos++;
   //no need to do anything for this segment, move on
   continue;
  }
  groupContains[segmentId]=true;
  groupArray[groupArrayMax++]=segmentId;
  if (segmentId==VSS) { groupFalse=true; continue; }
  if (segmentId==VCC) { groupTrue=true; continue; }

  foreach (transistor in segment.c1c2s)
  {
   if (transistor.on)
   {
    if (transistor.c1==segment) tableToAdd[maxPos++]=transistor.c2
    if (transistor.c2==segment) tableToAdd[maxPos++]=transistor.c1
   }
  }
 }
}
```

By these changes all recursive calls are eliminated as well as dealing with all dynamic collections. As this method is called hundreds of thousands of times per

second, any saving on either allocation of new object or additional method call will yield large performance benefits.

3.1.2. Introducing flat arrays on top of collections

Most of dynamic lists were replaced with static arrays. This applies for *groupList* in the original code replaced with *groupArray*. Also all static lists and hashmaps (for global transistors list, global segment list, gates list for a given segment, C1C2 list for a given segment, etc.) were augmented with static arrays. That helped eliminating any overhead on referencing to these lists in the critical parts of the simulation (like in *addSegmentToGroup()*). Most of the costly internal *Collection.get()* and *valueOf()* calls were removed.

3.2. Optimizations in the algorithm itself

Below are several recommendations for the algorithm itself.

3.2.1. Ahead of time group value calculation

One of improvements is based on the observation that as the group gets created (as part of the *addSegmentToGroup()* recursive method, or its refactored flat, iterative based version) – all the information that is needed to calculate the group value is already present. Therefore instead of performing a full iteration through the group as part of the *getGroupValue()* the value of the group can be set every time the segment is added to the group and memoized. Two additional boolean fields in the class are used for that:

```
boolean isGroupTrue
boolean isGroupFalse
```

At group creation they are both set to FALSE. Then whenever a segment is added in *addSegmentToGroup()* to the group, it's checked against GND and PWR and the above flags are set accordingly. Finally the *getGroupValue* takes a form like below:

```
getGroupValue(segmentGroup)
{
 if (isGroupFalse) return false;
 if (isGroupTrue) return true;
 if (group.contains(segment.state=pullup)) return true;
 return false;
}
```

It was observed that more than 50% of groups contain either GND or PWR. The above change allows then much faster group value evaluation in all these cases, without a need of iterating through the group.

3.2.2. Leveraging information from previous iteration

Part of the refactored, flat and iterative *addSegmentToGroup()* is a lookup boolean array *groupContains[]*. Every time new group is created, all positions in this lookup array shall be set to false. A naive, brute force approach would be to either iterate through the array for each group or to create the array from scratch. With the first approach there is obviously an overhead on iterating over MAX_TRANSISTORS elements (where in 6502 MAX_TRANSISTORS count reaches 3500). With the latter approach there is an overhead related to memory allocation for a brand new array and also CPU time consumed by the GC. What can be used alternatively is the *groupArray* and its *groupArrayMax* index. As groups are relatively small, iterating over previous group up to the previous *groupArrayMax* and setting the corresponding *groupContains[id]* entries to false will effectively prepare it for the next group.

3.2.3. Cutting method execution as early as possible

In several places of the original algorithm it was observed, that parts of the code shall not be executed at all. This is especially relevant for sections of the code that do not make any sense to execute for special segments, like VCC (power) or VSS (ground). While execution of the code does not have any impact on the actual simulation (programs run just fine), cutting the execution early helps in performance improvements. One of such places is highlighted below – with additional check added:

```
recalcSegment(segment)
{
 //there is no point to evaluate
 //ground (VSS) or power (VCC)
 if (segment == GND or PWR) return;

 segmentGroup = new List();
 addSegmentToGroup(segment);
 newState = getGroupValue(segmentGroup);
 foreach (segment in segmentGroup)
 {
  if (segment==VCC or segment==VSS) continue;
  if (segment.state!=newState)
  {
   Segment.state = newState;
```

```
    foreach (transistor where transistor.gate=segment)
    {
     if (transistor changes to on)
     {
      addToNextIterationList(tr.c1);
     }
     if (transistor changes to off)
     {
      addToNextIterationList(tr.c1);
      addToNextIterationList(tr.c2);
     }
    }
   }
  }
}
```

4. Improved implementation results

Once the above improvements are applied – significant speed gains are observed (see Figure 4).

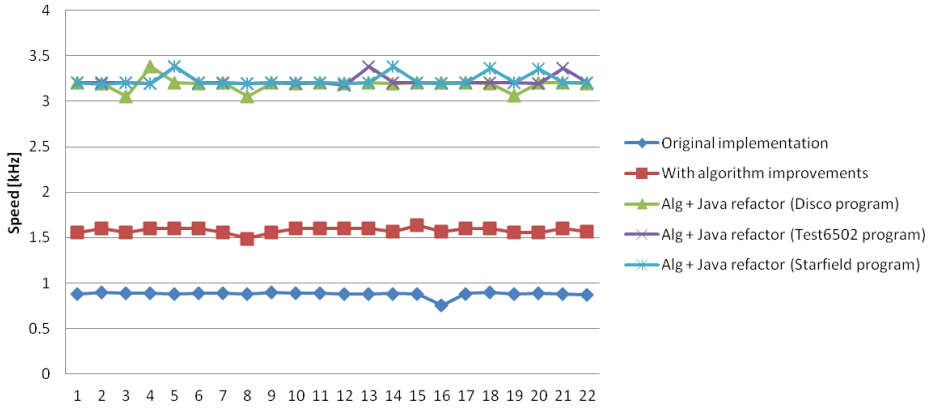

Figure 4. Simulation speed improvements.

The chart above shows a series of speed measurements taken on initial implementation (reaching 0.9 kHz), then once the algorithmic improvements are applied (1.6 kHz) and finally once the Java/JVM refactoring is implemented (∼3.2 kHz). The measurements were taken every full 1000 CPU cycles (so were measured over 20 000 CPU cycles). Final implementation (with all optimisations) was tested against 3 different 6502 benchmark programs (Disco [13], Test6502 – as originally showcased in [5] and Starfield [13]), to avoid optimising for particular sequence of machine code commands. The results of these tests show consistent improvements in speed. That puts this transistor level simulator implementation in range of higher, RTL level simulators.

5. Further work discussion

It was presented, that applying some point optimisations on the implementation language level and optimising the algorithm in some carefully selected hotspots allowed increasing significantly the simulation speed. It shall be noted however that there is still potential for further speed improvements by exploiting the memoization especially in the area of group creation. Groups ('splats' of locally interconnected segments, see Figure 5) are small in size. The reach of the group depends on the on/off state of transistors close to the group.

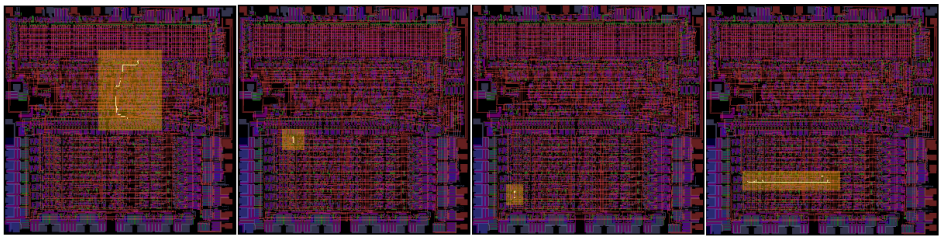

Figure 5. Independent group splats of interconnected transistors.

On the figure above several subsequent evaluated groups are highlighted (segments in white, yellow bounding box for highlight purposes only). It is worth to explore the option of group creation lookup tables (where groups for a given segment would be memoized, and fetched from the lookup table based on the state combination of nearby transistors). Another area of improvement may be in enabling the parallelism of the algorithm. Currently it is strongly sequential, single threaded, but visualising and observing how subsequent groups are evaluated often in far, independent areas of the chip, indicates there is a feasible option to levelize the transistors, and make some calculations in parallel.

6. References

[1] Matthews B., Shaon A., Bicarregui J., Jones C., *A framework for software preservation.* The International Journal of Digital Curation, 2010, 5(1), pp. 91–105.

[2] Trevor O., *Preserving.exe: Towards a National Strategy for Software Preservation. NDIIPP.* http://www.digitalpreservation.gov/multimedia/documents/PreservingEXE_report_final101813.pdf, 2013 [Accessed 7-July-2014].

[3] van der Hoeven J., van Wijngaarden H., Verdegem R., Slats J., *Emulation – a viable preservation strategy.* http://dioscuri.sourceforge.net/docs/Emulation_report_KBNA_ 2005_en.pdf, 2005 [Accessed 5-March-2015].

[4] Adam, *DICE - Digital Integrated Circuit Emulator.* http://adamulation.blogspot.com/, 2012 [Accessed 5-July-2014].

[5] James G., Silverman B., Silverman B., Visualizing a classic cpu in action: the 6502. In: *SIGGRAPH Talks*, ACM, 2010.

[6] Dunkels A., *Contiki Project.* http://www.contiki-os.org/, [Accessed 10-March-2015].

[7] Dunkels A., Gronvall B., Voigt T., Contiki - a lightweight and flexible operating system for tiny networked sensors. In: *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, IEEE, 2004.

[8] S. E., *6502 Simulating in Real Time on FPGA.* http://visual6502.org/wiki/index.php?title=6502_-_simulating_in_real_time_on_an_FPGA, 2011 [Accessed 5-March-2014].

[9] Van Weeren D., *Amiga Minimig OpenSource Project.* https://code.google.com/p/minimig/, 2010 [Accessed 8-March-2015].

[10] Alles M., Amsdon L., Aschwanden F., *Atari ColdFire Project.* http://acp.atari.org, 2009 [Accessed 8-March-2015].

[11] Wojtowicz T., *Visualizing cpu microarchitecture.* Schedae Informaticae, 2015, 23.

[12] James G., Silverman B., Silverman B., *Visual 6502 CPU simulator.* http://www.visual6502.com/, 2011 [Accessed 5-July-2014].

[13] Soreng S., *6502 Compiler and Emulator in Javascript.* http://www.6502asm.com/, 2009 [Accessed 28-March-2015].