Exploratory Efforts to Manage Power-Aware Memories using Software Generated Hints

Mohammad Hammoud and Rami Melhem
Department of Computer Science
University of Pittsburgh

Abstract

This report presents our exploratory efforts for managing main memory power-aware chips. Current state-of-the-art power-aware DRAM chips offer various power modes (*active, standby, nap, and powerdown*) in order to provide a potential to limit power consumption in the face of increasing demand for performance. Our goal in this study is to utilize and exploit these various power modes for the most effective main memory power management under software control in response to workloads becoming increasingly memory-intensive and data-centric.

I. Introduction

This report presents our exploratory efforts to minimize energy consumption of memory by power mode control at memory bank granularity. A software approach has been adopted and experimental studies have been conducted to explore the potential benefits of such an approach. A fully functional detailed execution driven simulator, Simics, has been utilized to perform the experimentations. A power-aware memory module that contains a number of memory nodes organized as an array of banks has been constructed and loaded into Simics. The module supports multiple power modes and the ability to initiate a transition from one to the other. Each power mode is characterized by its power consumption and the time it takes to transition back to the active mode (resynchronization cost). The report bases the experimental results on a simulated machine both, with and without the presence of a cache hierarchy. System with caches is what actually counts, but masks the actual characteristics of the application. Hence, and to reflect the effect of the OS due to page allocation policies, a system without caches has been also simulated.

A study has been conducted to scrutinize memory traffic/accesses generated by comprehensive applications (started with TPC-C then focused on the Java application server benchmark, SPECJBB, beside some in-house Java mini-benchmarks) to the constructed banked memory architecture. Java applications have been the focus because they stress the memory system more than traditional programs. For instance, *bytecodes* are treated as data and need to be fetched from memory for interpretation or JIT-compilation. Moreover, JVM features, such as garbage collection, make Java executions much more memory-intensive than normal programs.

After constructing a power-aware memory module and loading it into Simics, special directives (magic instructions) have been instrumented within the benchmarks source codes (SPECJBB and the in-house Java mini-benchmarks) so as to force some simulation actions. As such, this allowed us to identify the different transaction types generated by the SPECJBB benchmark. SPECJBB models a wholesale company. A spawned thread in the SPECJBB benchmark represents an active user posting transaction requests within the warehouse. The goal was to track a certain memory access pattern in correlation with the generated transaction types. Such an aspired correspondence was aimed to be considered as a sort of hint that can be provided by the compiler so as to judiciously assist in manipulating power modes of the memory banks.

Finally, a study has been conducted also to identify energy bottlenecks for different software components of the execution, such as the application itself isolated from any other incorporated component (i.e. JVM), the JVM, and the OS. An in-house very simple Java benchmark has been run on top of a JVM so as to isolate the energy effect of the JVM and the OS.

The rest of the report is organized as follows. Section 2 states the goals and contributions of the investigation. Section 3 delves onto the experimental platform and explains it in detail. Details of the experimental results are presented in section 4, and section 5 concludes.

II. Goals and Contributions of the Investigation:

Given the objective of minimizing energy consumption of memory by power mode control at memory bank granularity, this report sets the following goals:

- Discover hints that the compiler/JVM can provide so as to detect/predict module idleness and perform transitions accordingly (i.e. Hints about types of transactions)
- Study the interaction between memory allocators of OS or JVM, memory controller, and application behavior so as to understand the involved roles in determining the memory access patterns.

The contributions of the report are threefold:

- No specific correlation detected between transaction types and memory access patterns. Therefore, compiler hints based on transaction types are not of real use.
- The roles of the OS and JVM memory allocators and garbage collector (GC) are crucial in determining the memory access patterns.
- GC is a crucial part of the JVM and greatly important for energy-constrained memory architectures.

III. Experimental Platform:

A. Power-Aware Memory Module

Simics allows us to observe and sometimes modify the behavior of the transactions that go through the memory system. Simics API provides some interfaces (i.e. timing-model and snoop memory) that can be implemented in a module written in C or Python. Such a module can be added to a Simics installation to

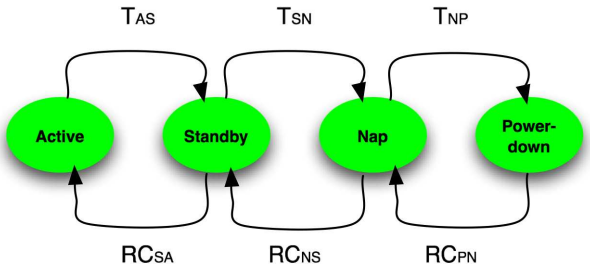

Fig. 1. Operating Modes.

essentially extend its functionality. A memory module that contains a number of memory nodes organized as an array of banks each of which can be independently set to a different power state has been developed and loaded into Simics. The number of memory banks can be set within the memory module code. The conventional memory interleaving scheme is utilized to allocate addressed data blocks to memory banks. The size of the interleaved data blocks can be set to different values: a word, a cache line, multiple cache lines, a page, or multiple pages. Four power states are offered (listed in decreasing order by power dissipation but increasing access time): active, standby, nap, and powerdown. A bank must be in the active mode when accessed for a read or write transaction. A bank not servicing memory requests can be in any of the lower power modes. Power modes are dynamically determined. If a bank is not accessed for a threshold amount of time, it transitions to the next lower power mode. When accessed again a performance penalty (resynchronization cost) is added for transitioning it back to the active mode so as to service the request. Figure 1 pictorially depicts the transitions and resynchronization costs that can occur between the four different offered power states. T_{AS}, T_{SN}, and T_{NP} are the time thresholds for a bank to transition from active to standby, standby to nap, and nap to powerdown respectively. RC_{SA}, RC_{NS}, and RC_{PN} are the resynchronization costs penalized when transitioning from standby to active, nap to standby, and powerdown to nap respectively. The time that each bank spends on a certain power mode can be measured and the total energy consumption can be accordingly computed. Execution time and frequency of accesses to each bank for each power mode can be furthermore reported.

B. Special Directives to Force Simulation Actions

A module written in C or Python and loaded into Simics may want to react to certain Simics events. This may both be events related to the simulated machine, like processor exceptions or control register writes, and other kinds of events like Simics stopping the simulation and returning to the command line. Simics such events are referred to as *haps*. To react to a hap, a module can register callback functions to the hap. For each simulated processor architecture, a special no-operation has been chosen to be referred to as a magic instruction for the simulator. When Simics executes such an instruction it triggers an already defined hap known as the Core_Magic_Instruction and, consecutively, calls all the callbacks functions (handlers)

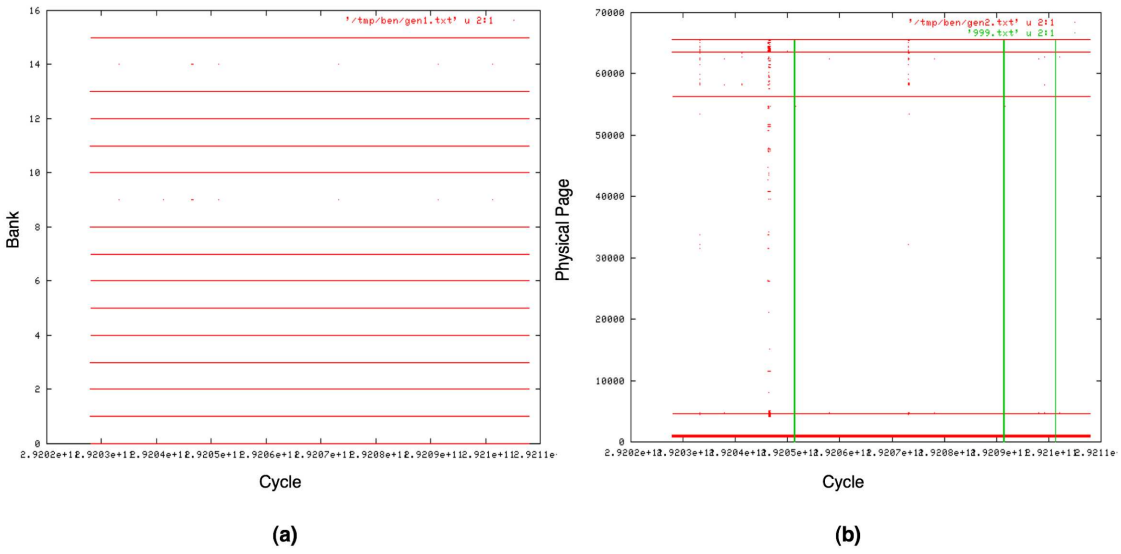

Fig. 2. Results with the Simulated Machine Being Idle (No Caches and No Benchmark Running on Top of the Machine).

registered on this hap. An immediate value can be encoded in the magic instruction and when the hap is triggered the value is passed as an argument to the hap handlers. For instance, for the SPECJBB benchmark, we identified the different transaction types within the source code and then encoded a value for each specified type within the magic instructions instrumented inside the code. When Simics encounters any of these instrumented magic instructions, it triggers the Core_Magic_Instructin hap and an encoded value denoting a specific transaction type is simply passed to the associated hap handlers registered by our memory module that in return characterizes the transaction type.

C. Configuration:

Our simulated machine has a single Pentium 4 processor, runs Red Hat Linux 7.3 operating system, and has a total of 256 MB of memory. The basic data block is a page consisting of 4K Bytes. L1 and L2 caches are pinned to the system when required. L1 cache is 16 KB I/D and 4-way set associative with 1 cycle access time and 64 byte line. L2 cache is 4MB and 16-way set associative with 6 cycles access time and 64 byte line.

IV. Simulation Results

V. Details of the Experimental Results

A. The Simulated Machine Being Idle

The simulated machine has been run firstly without any cache hierarchy and with no benchmark running on top of it. This is to reflect upon the memory traffic/accesses generated by the OS due to page allocation policies. Figure 2(a) depicts the memory traffic experienced to the constructed memory architecture over

```
//Jvm startup
MagicBreak.magic(0);   //green line
for (int i=0; i<6250000; i++);
MagicBreak.magic(0);   //blue line
//Jvm cleanup
//the machine is idle
```

Fig. 3. An In-house Java Mini-Benchmark.

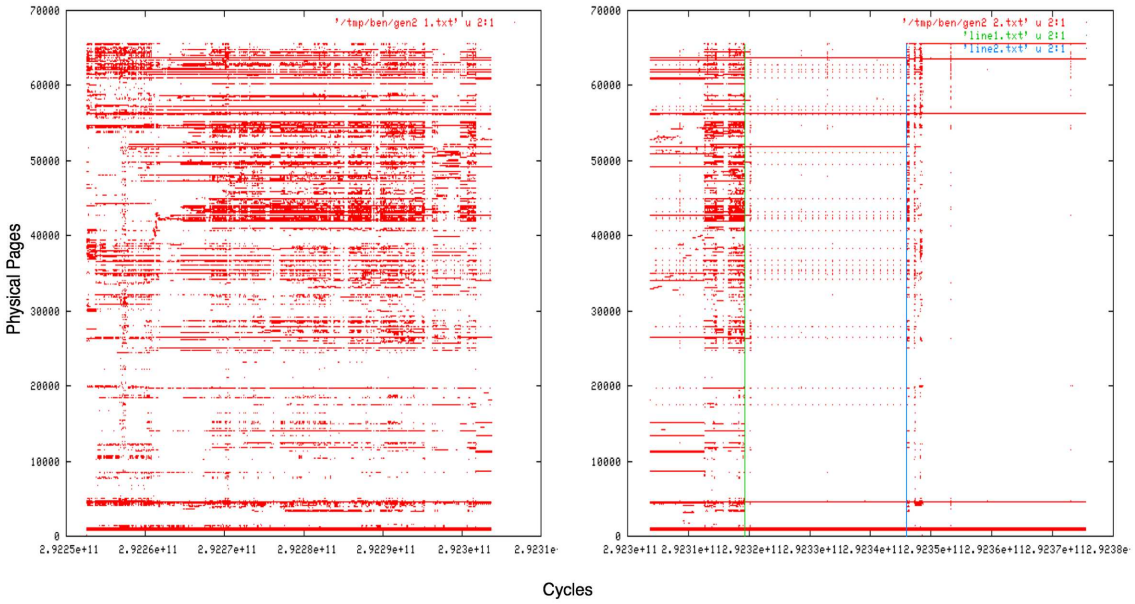

Fig. 4. Results for Memory Access Pattern During JVM Initialization Phase.

time at bank granularity. Though the machine is idle, clearly the OS touches almost every bank. Figure 2(b) further depicts the memory traffic to the underlying memory architecture over time but at the physical page granularity. It has been observed that all memory accesses are from the kernel and 0.4% of pages have been touched. In essence, the kernel periodically *scans* the physical memory.

B. Memory Access Pattern during JVM Initialization:

The mini-benchmark program illustrated in Figure 3 has been run on top of the simulated machine. Via running such a simple benchmark, memory access patterns will be easy to learn. Figure 4 depicts the memory traffic to the constructed memory architecture over time at the physical page granularity. The figure demonstrates that the memory accesses during JVM startup are essentially a mess.

```
MagicBreak.magic(0);    //beginning of the 1st graph
int a[]= new int[6250000];  //25MB
MagicBreak.magic(0);    //green line of the 2nd graph
for (int i=0; i<6250000; i++) a[i]=1;
MagicBreak.magic(0);    //green line of the 4th graph
int b[] = new int[6250000];
MagicBreak.magic(0);    //green line of the 5th graph
for (int i=0; i<6250000; i++) b[i]=1;
MagicBreak.magic(0);    //end of the last graph
```

Fig. 5. An In-house Java Mini-Benchmark.

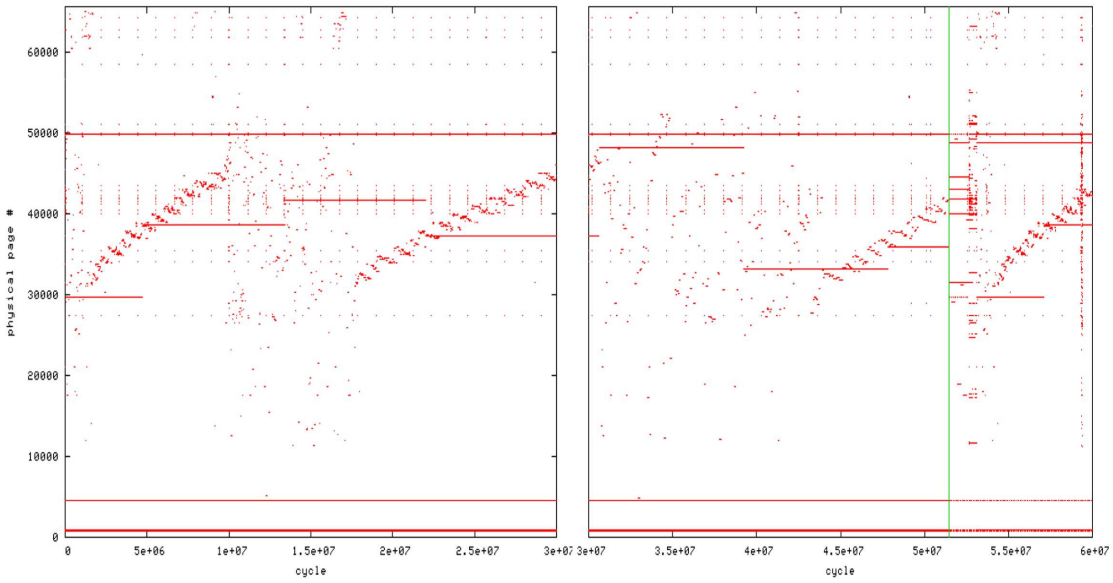

Fig. 6. Results for Memory Access Pattern During Java Objects Allocation.

C. Memory Access Pattern upon Java Objects Allocations:

The mini-benchmark program illustrated in Figure 5 has been simulated. Figure 6 depicts the memory traffic over time at the physical page granularity. Array **a** seemed to occupy much larger than 25 MB of the physical memory; however, in reality it didn't. Furthermore, the GC seemed to have collected the memory of **a** before allocating memory for **b**; however, also in reality it didn't.

D. Memory Access Pattern of the Garbage Collector:

The mini-benchmark program illustrated in Figure 7 has been run on top of the simulated machine. Figure 8 depicts the memory accesses over time at the physical page granularity. The memory accesses during garbage collection, as demonstrated, are messy. Furthermore, though gc() doesn't collect memory, it

```
public class TestMagicBreak{
     int data[] = new int[2500000];  //10MB

     public static void main(String args[]){
          TestMagicBreak b;

          System.out.println(Runtime.getRuntime().freeMemory());
          MagicBreak.magic(0);
          b = new TestMagicBreak();
          MagicBreak.magic(0);
          System.out.println(Runtime.getRuntime().freeMemory());
          for (int i=0; i<2500000; i++) b.data[i]=1;

          b = new TestMagicBreak();
          System.out.println(Runtime.getRuntime().freeMemory());
          for (int i=0; i<2500000; i++) b.data[i]=1;

          b = new TestMagicBreak();
          System.out.println(Runtime.getRuntime().freeMemory());
          for (int i=0; i<2500000; i++) b.data[i]=1;

          MagicBreak.magic(0);
          System.gc();
          MagicBreak.magic(0);
          System.out.println("gc");
          System.out.println(Runtime.getRuntime().freeMemory());

          TestMagicBreak c = new TestMagicBreak();
          MagicBreak.magic(0);
          System.out.println(Runtime.getRuntime().freeMemory());
          for (int i=0; i<2500000; i++) c.data[i]=1;
          MagicBreak.magic(0);
          for (int i=0; i<2500000; i++) b.data[i]=1;

          MagicBreak.magic(0);
     }
}
```

Fig. 7. An In-house Java Mini-Benchmark.

still generates a great deal of memory traffic. Finally, a *decreasing access pattern* has been observed in the 6^{th} plot of the figure.

E. Correlation between Memory Access Patterns and Transaction Types with No Cache Hierarchy:

The SPECJBB benchmark has been run on top of the simulated machine *with no* cache hierarchy in an objective to detect correlations between memory access patterns and transaction types. Figure 9 portrays the memory accesses over time at bank granularity for the following transaction type sequences respectively: [0-> 5-> 0-> 1-> 5-> 0-> 0-> 1-> 4->], [0-> 5-> 0-> 1-> 3-> 1-> 1->], [0-> 1-> 1-> 2-> 5->

0-> 5-> 1-> 0-> 5-> 0-> 0-> 0->], [1-> 0-> 1->]. Figure 10 portrays the accesses over time at the physical page granularity while Figure 11 demonstrated the traffic over time at the virtual page granularity and both for the same above mentioned transaction type sequences. Finally Figure 12 shows the access frequency to different banks for the different transaction types. No specific correlation has been detected between transaction types and memory access patterns.

F. Correlation between Memory Access Patterns and Transaction Types with Cache Hierarchy:

The SPECJBB benchmark has been run on top of the simulated machine *with* cache hierarchy this time. Figure 13 portrays the memory accesses over time at bank granularity for the following transaction type sequences respectively: [5-> 0-> 0-> 1-> 0-> 1-> 0-> 0-> 0-> 1-> 1->], [1-> 1-> 0-> 0-> 0-> 1-> 1-> 0-> 4-> 0-> 0-> 5->], [1-> 2-> 5-> 1-> 3-> 0->], [5-> 0-> 5->]. Figure 14 portrays the accesses over time at the physical page granularity while Figure 15 demonstrated the traffic over time at the virtual page granularity and both for the same above mentioned transaction type sequences. Finally Figure 16 shows the access frequency to different banks for the different transaction types. No specific correlation has been detected between transaction types and memory access patterns.

VI. Conclusion and Future Directions

This study revealed the importance of the role of the JVM in determining the memory access patterns. A main component of the JVM is the garbage collector (GC) that is responsible for automatic reclamation of heap allocated storage. GC may produce a large memory overhead that can essentially result in large energy consumption. Aspects of the GC (i.e. Mark & Sweep and Compaction Phases) can be tuned up at runtime to effectively mitigate the resultant overhead.

The compaction phase (CP) targets mainly the out-of-memory exceptions problem. Upon object allocation, if there is no enough heap space, the CP can interfere to rectify the situation via enlarging the heap free list by sliding live objects to the bottom (or top) of the heap memory area. This CP phase can effectively be exploited to move objects around banks so as to reduce energy. For example we need not wait for an out-of-memory exception to trigger the CP phase. Triggering the CP phase rarely and just because of a generated exception can simply bring up a fragmentation problem. This problem means more objects scattered over the memory and consequently more banks being active. The CP phase can in fact be controlled with energy consumption considerations so as to avoid any fragmentation problem.

Another critical issue is to when to trigger the Mark and Sweep (MS) phase of the GC. Essentially, the MS phase shouldn't be triggered leisurely. For instance, a bank completely full of garbage objects and kept on an active mode for a period time, T, means that it was wasting energy for T time. The MS phase can be exploited to reduce T by starting garbage collection at early stages. The MS and CP phases can further cooperate to free up more banks. The GC can fundamentally be made aware of the underlying main memory architecture (i.e. banked memory) to track the memory banks that are inactive. An inactive bank

might indicate that it contains objects that are garbage. A hint could be accordingly generated for the MS phase to simply start up.

To summarize, our future directions are as follows:

- To study the interaction of the GC with our banked memory architecture and see how its different components/phases can be tuned up appropriately to save energy.
- To propose mechanisms to tune up GC phases for energy consumption reasons.
- To study the sensitivity of the GC to the memory architecture and different memory patterns exposed by different java applications.

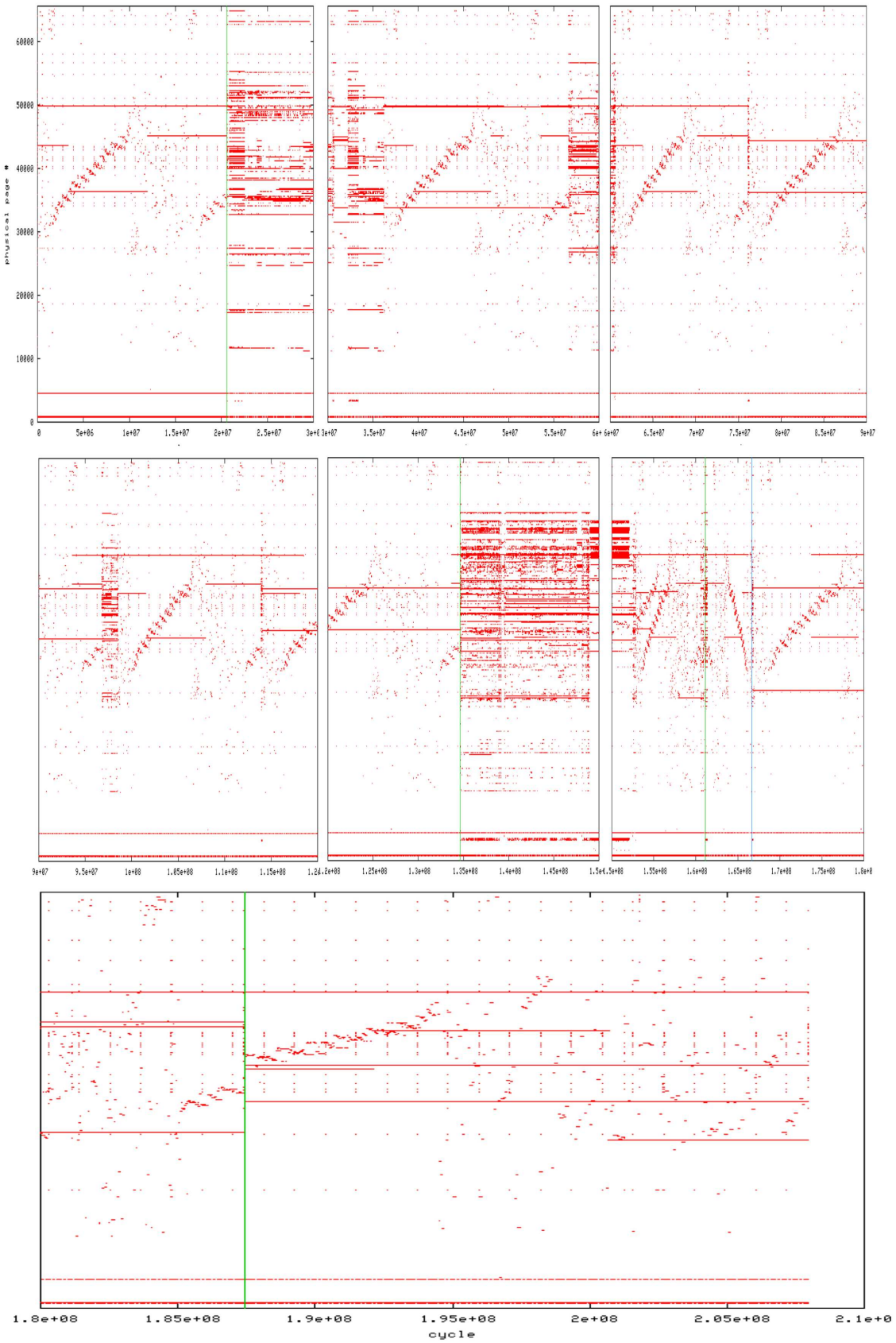

Fig. 8. Results for the Memory Access Pattern of the Garbage Collector.

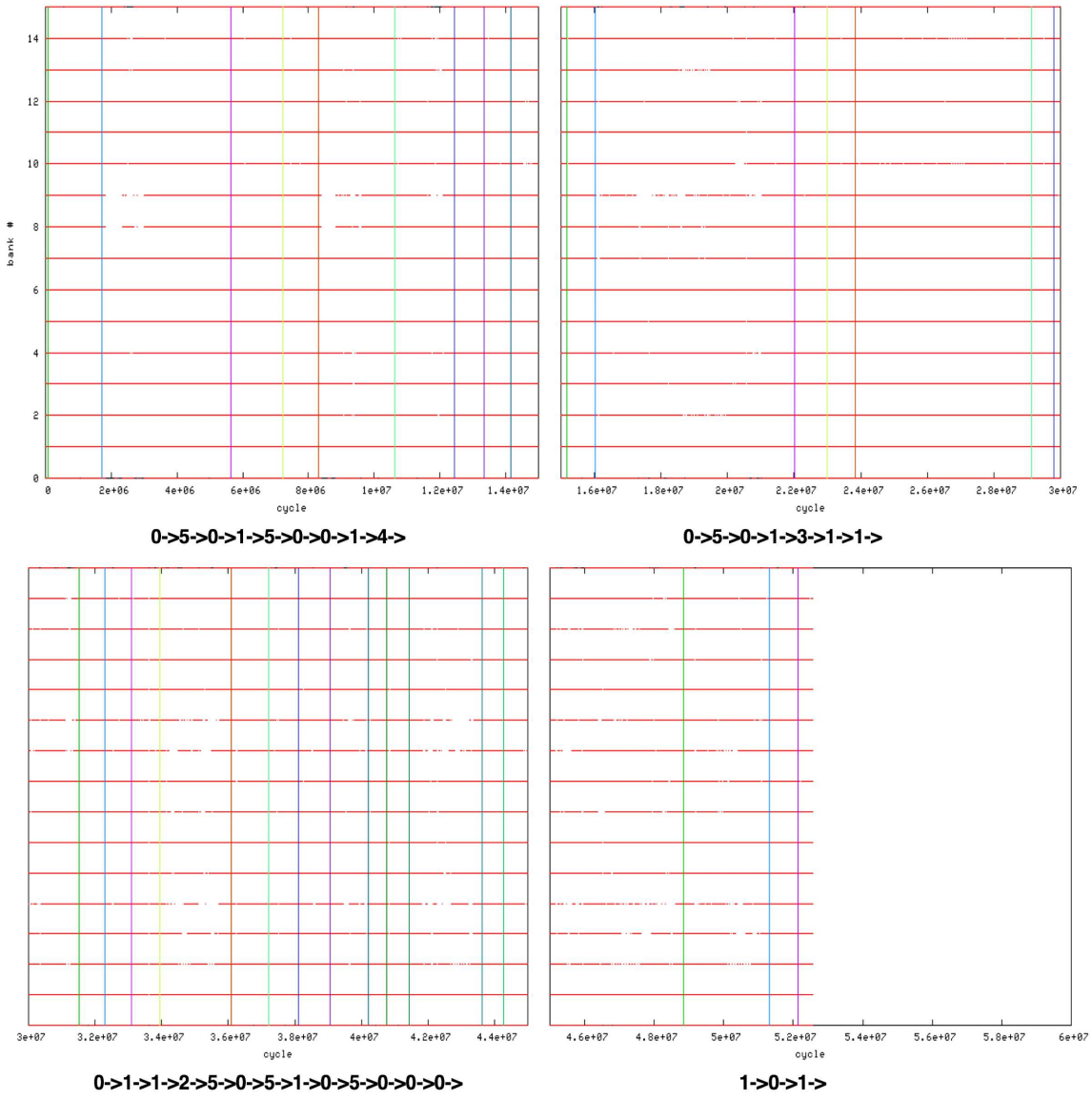

Fig. 9. Results for the SpecJBB Memory Accesses Over Time to the Memory Banks (No Cache).

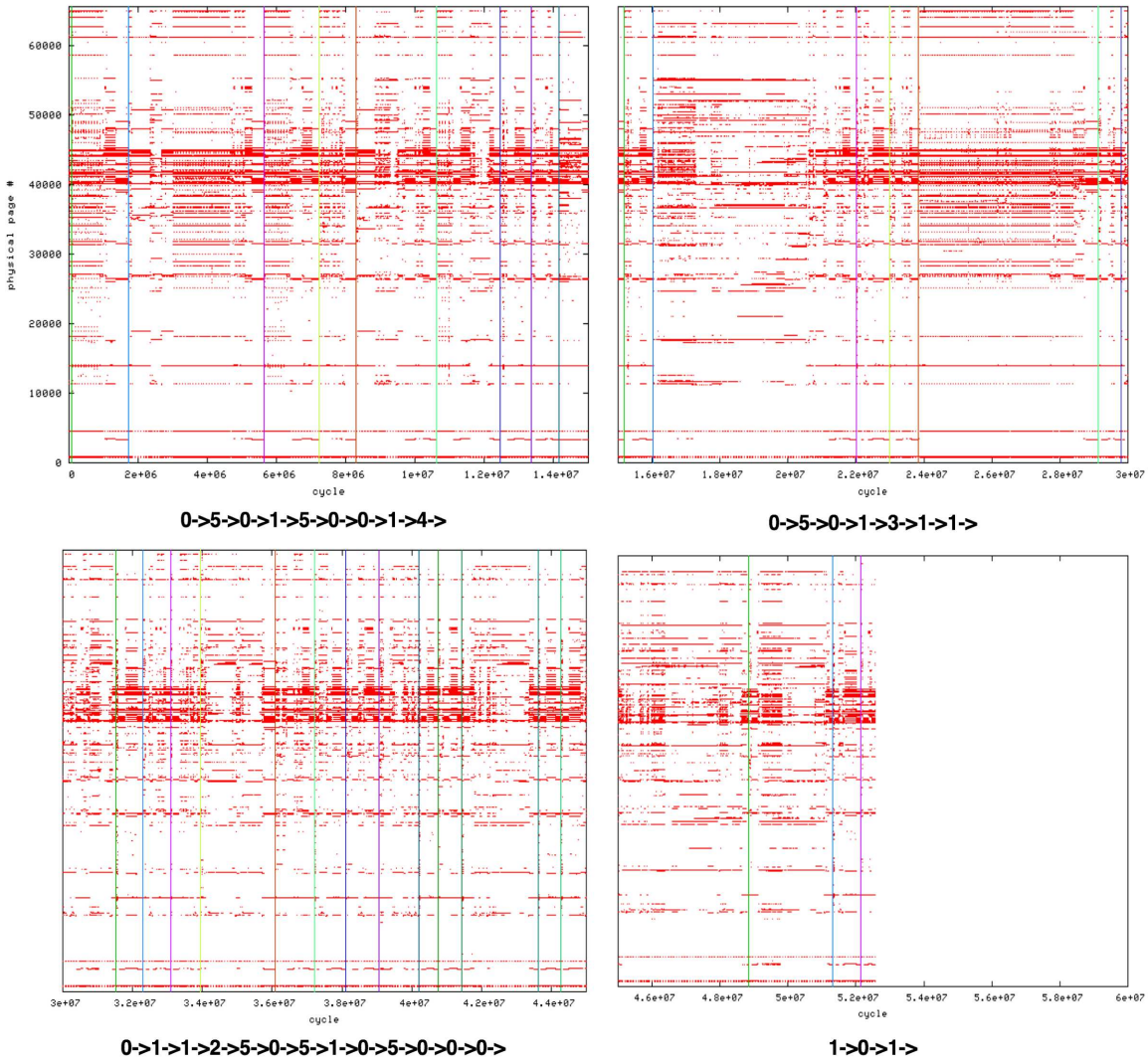

Fig. 10. Results for the SpecJBB Memory Accesses Over Time to the Physical Pages (No Cache).

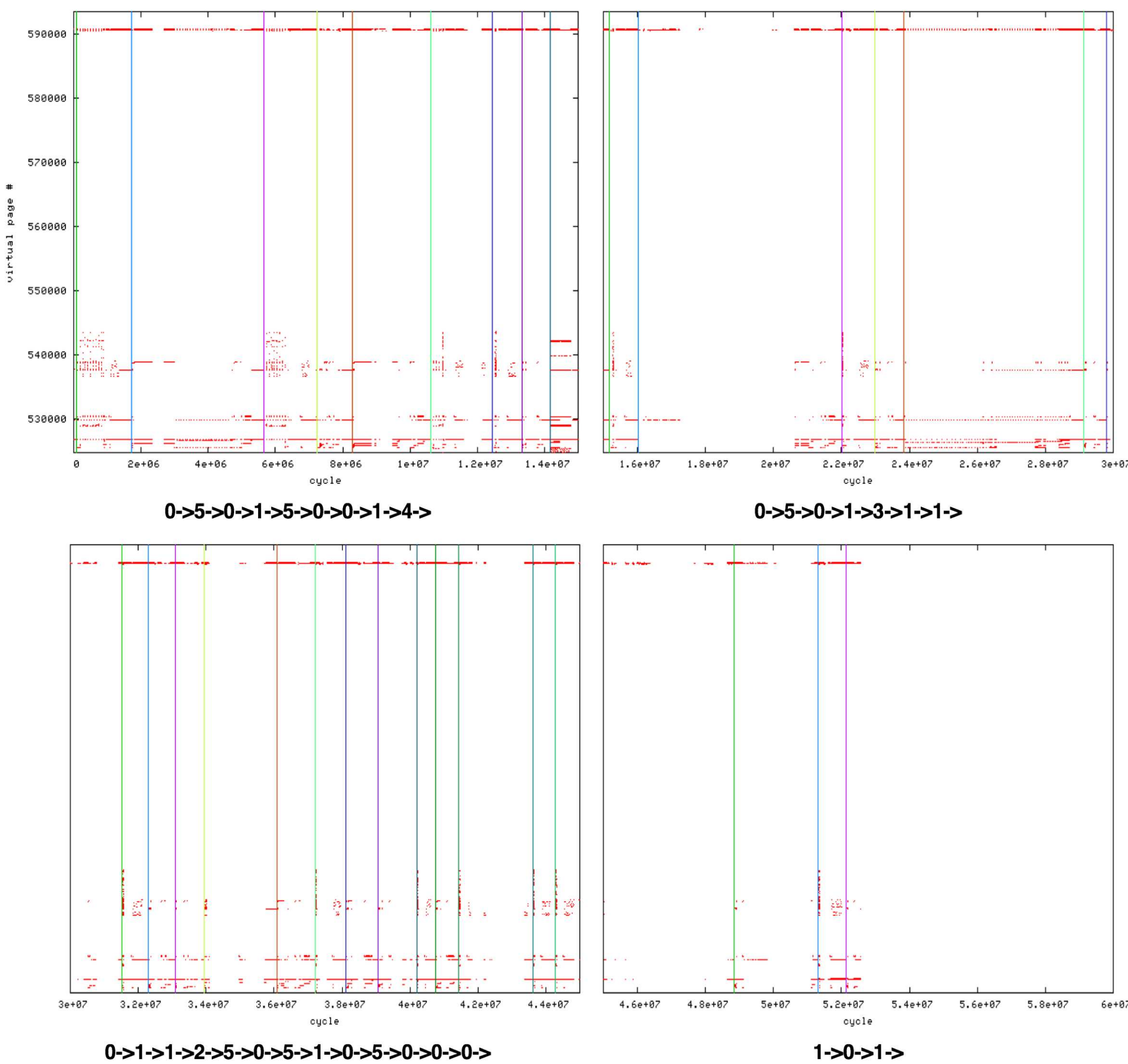

Fig. 11. Results for the SpecJBB Memory Accesses Over Time to the Virtual Pages (No Cache).

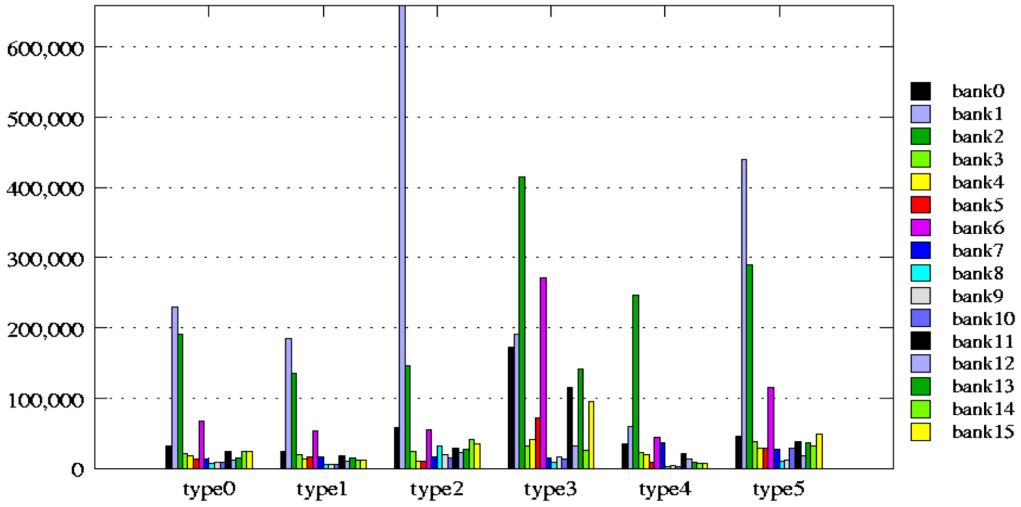

Fig. 12. Results for the Access Frequency to Different Memory Banks for Different SpecJBB Transaction Types (No Cache).

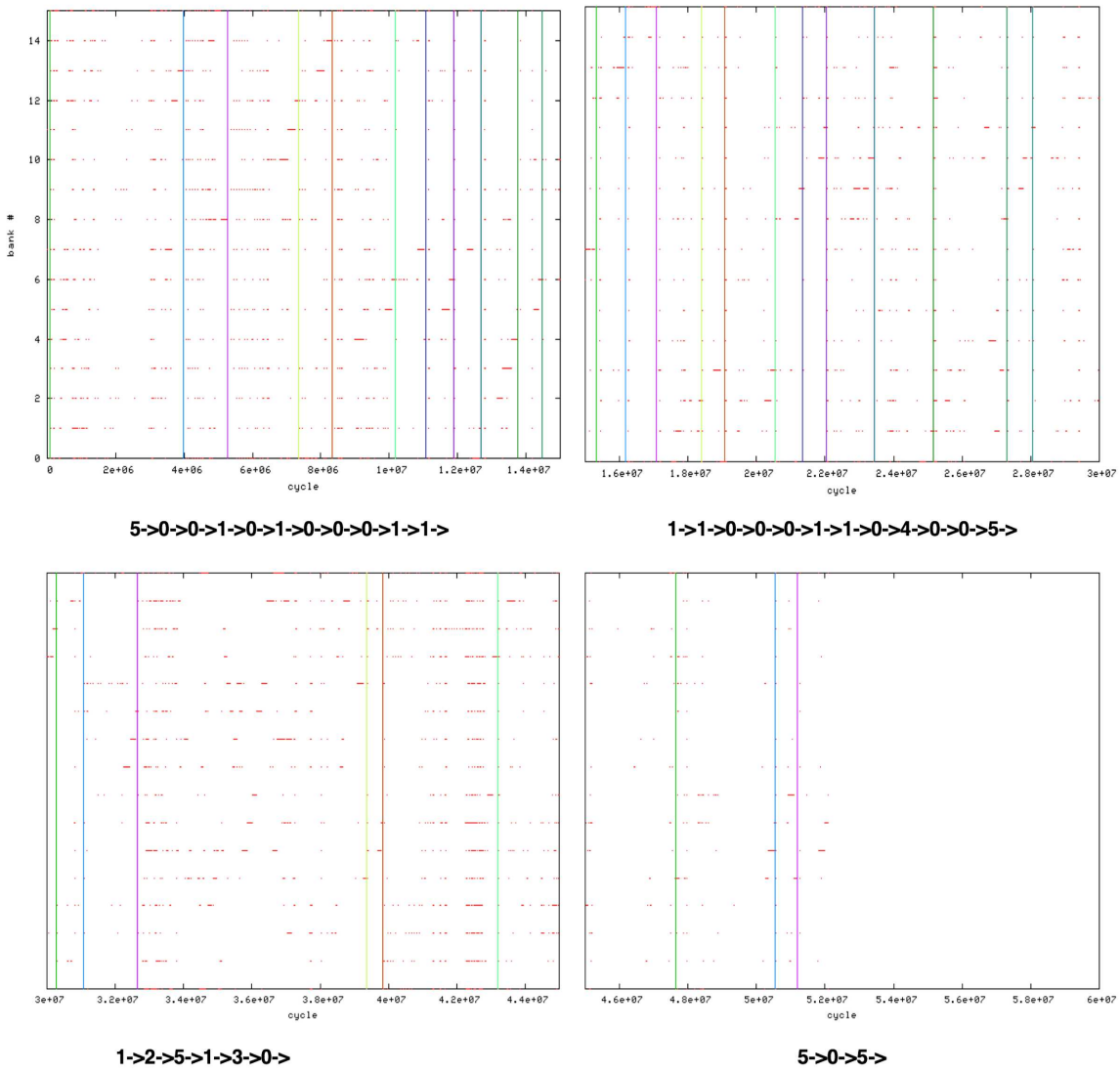

Fig. 13. Results for the SpecJBB Memory Accesses Over Time to the Memory Banks (With Cache).

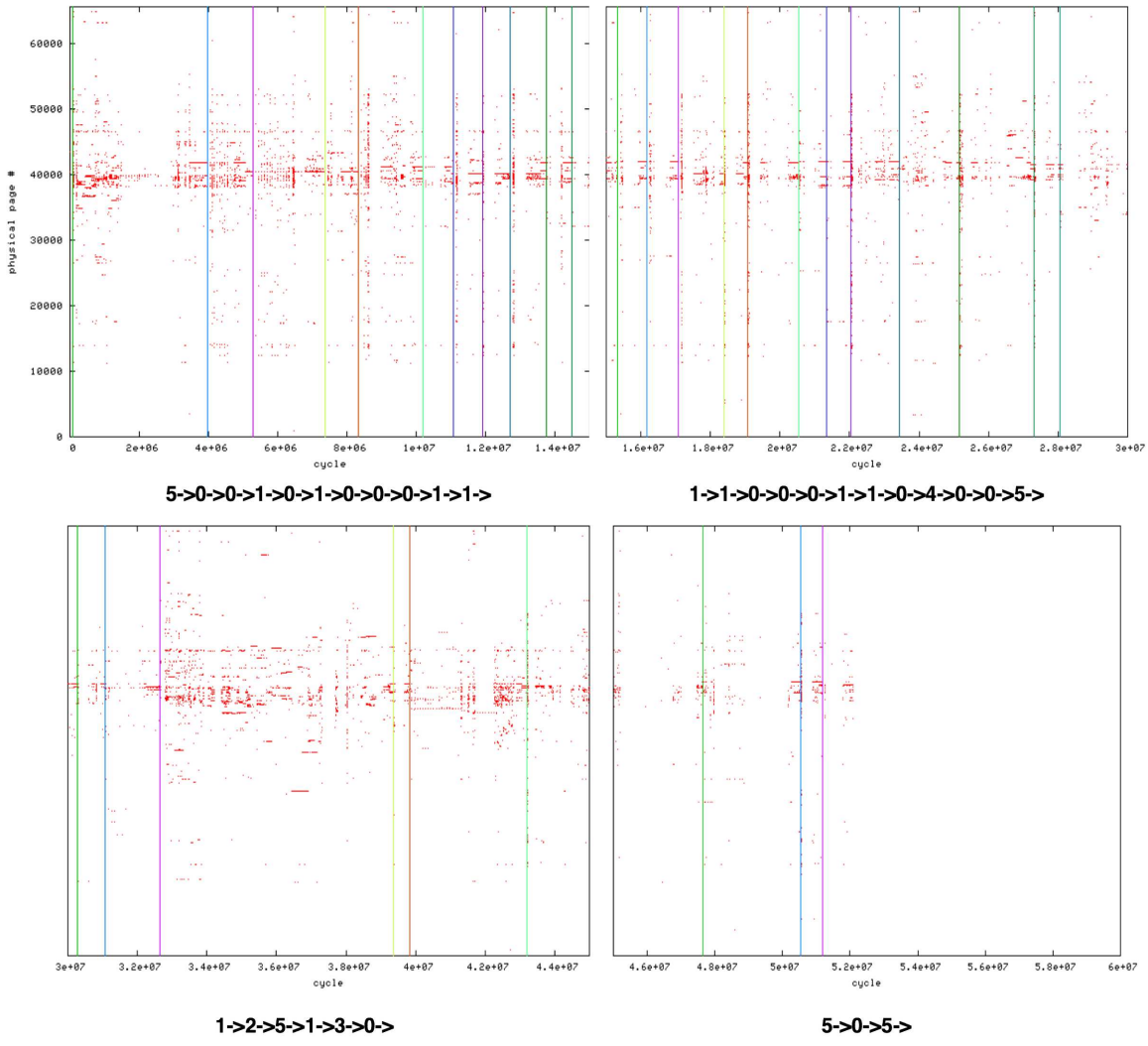

Fig. 14. Results for the SpecJBB Memory Accesses Over Time to the Physical Pages (With Cache).

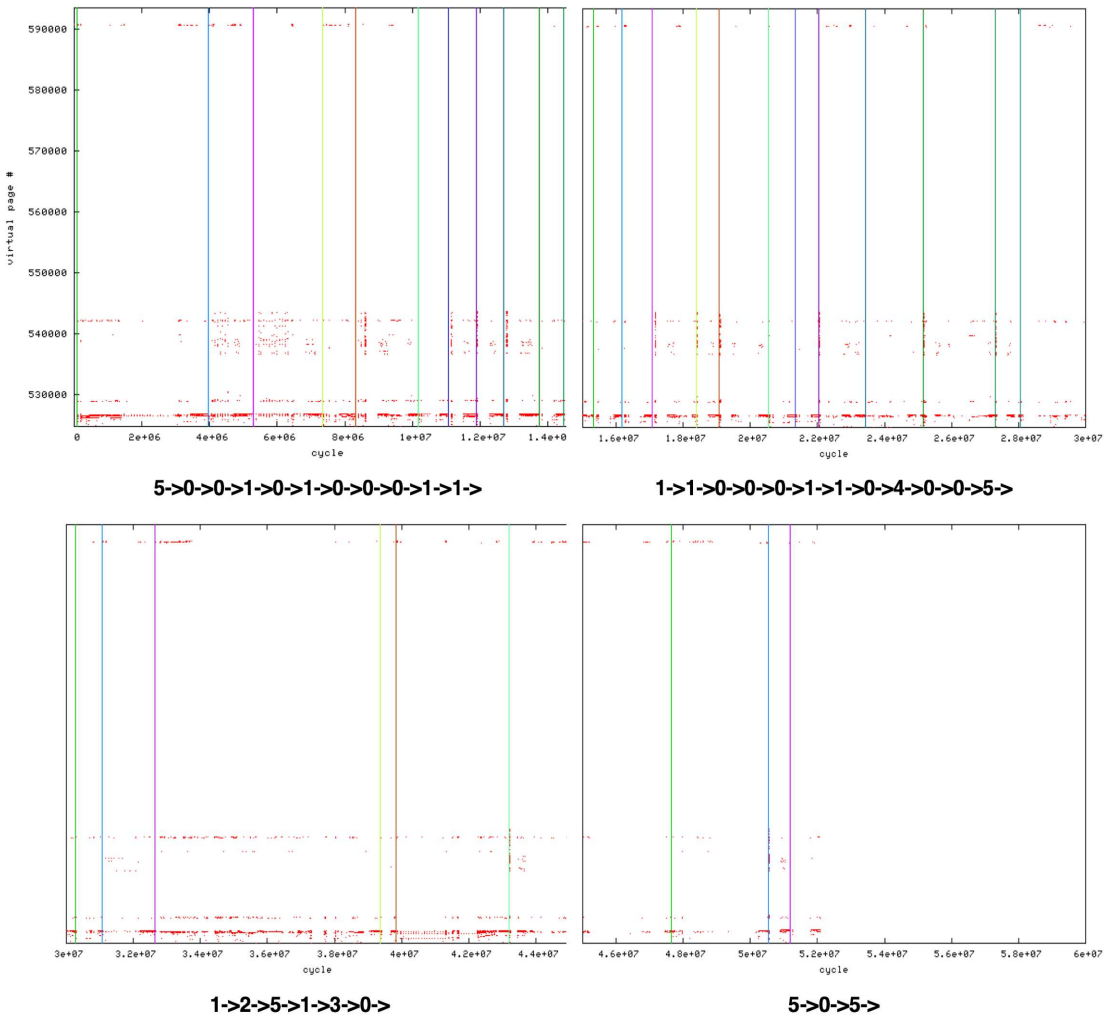

Fig. 15. Results for the SpecJBB Memory Accesses Over Time to the Virtual Pages (With Cache).

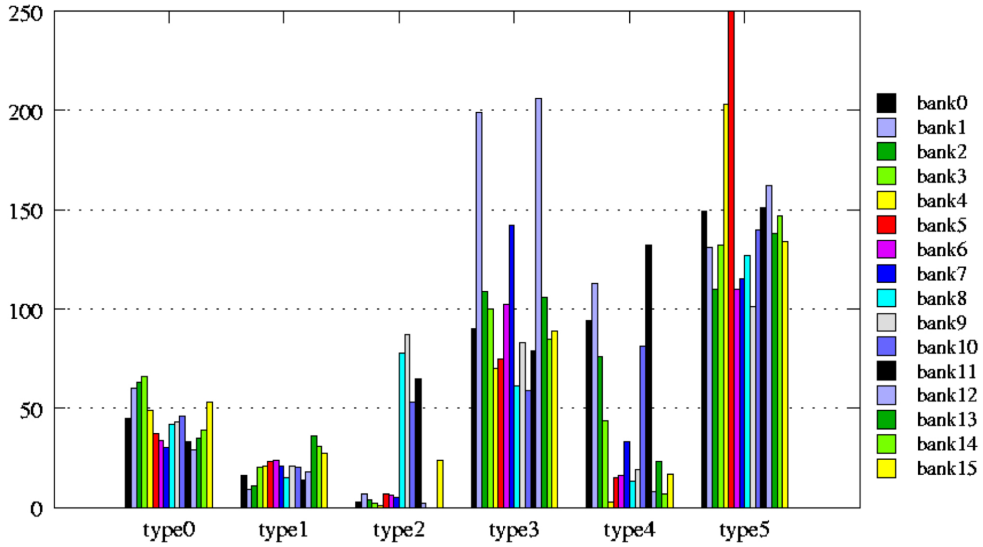

Fig. 16. Results for the Access Frequency to Different Memory Banks for Different SpecJBB Transaction Types (With Cache).