Using Field Access Frequency to Optimize Layout of Objects in the JVM

Taees Eimouri[1], Kenneth B. Kent[2], Aleksandar Micic[3], Karl Taylor[4]
[1,2]University of New Brunswick, Faculty of Computer Science, Fredericton, Canada
[3,4]IBM, Ottawa, Canada
{teimouri,ken}@unb.ca,{aleksandar_micic,taylor}@ca.ibm.ca

ABSTRACT

Increasing spatial locality of data can alleviate the gap between memory latency and processor speed. *Structure layout optimization* is one way to improve spatial locality, and consequently improve runtime performance, by rearranging fields inside objects. This research examines modifying IBM's JVM with the ability to reorder fields inside Java objects from access frequency information (hotness) in the presence of storage optimization.

INTRODUCTION

Structure layout optimization is performed based on:

- ***Hotness***: the total number of accesses to a particular field.
- ***Affinity***: fields accessed close to each other in time.

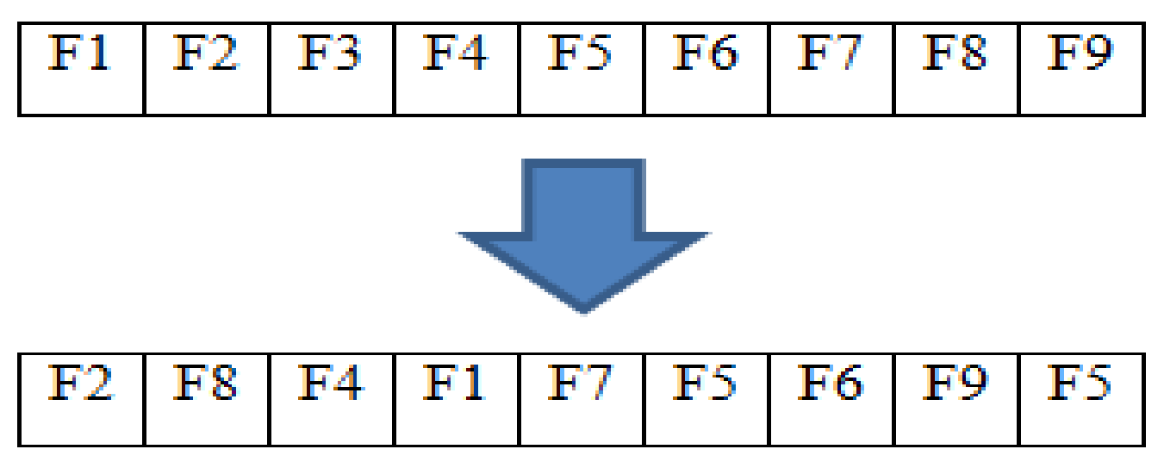


Figure 1: Field reordering using hotness or affinity.

HOTNESS ANALYSIS

Step 1: gathering information about particular classes and their non-static fields in a profiling run.

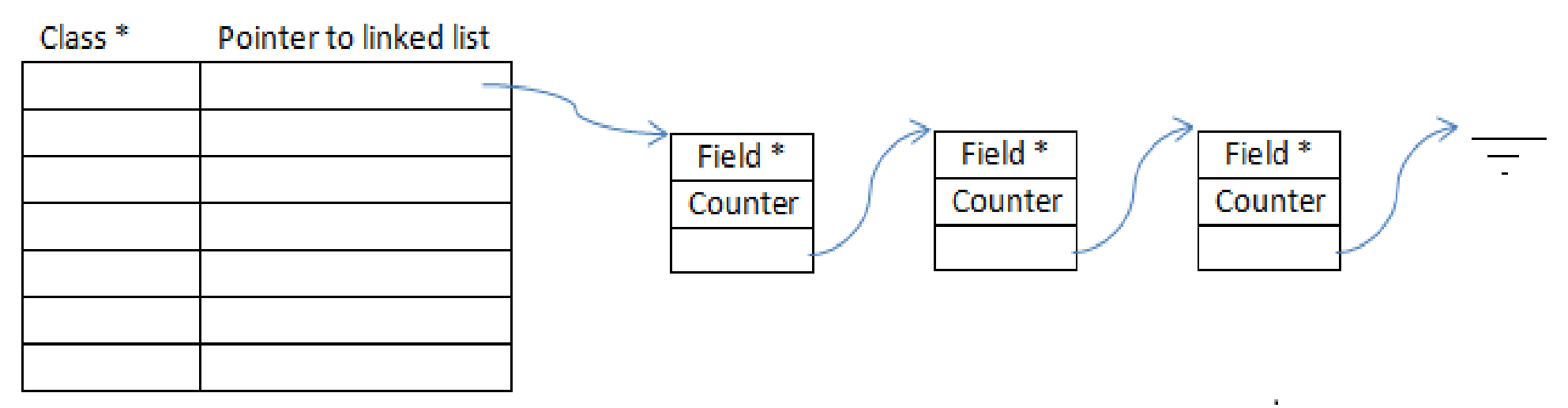


Figure 2: A hash table used in profiling.

HEURISTICS

IBM's JVM has its own layout scheme for objects that is well optimized from a memory footprint point of view.

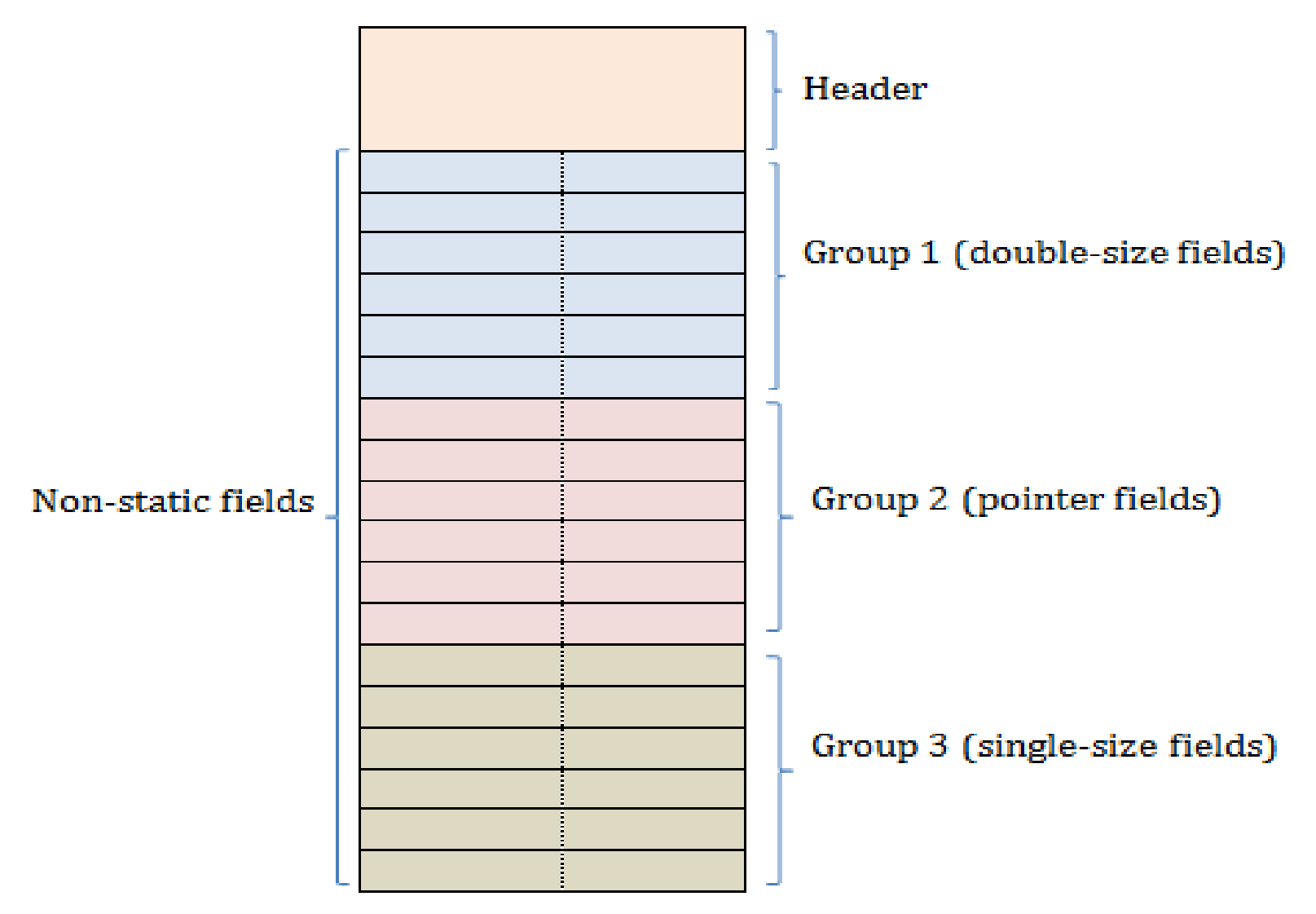


Figure 3: One sample object's layout in IBM's JVM.

Step 2: perform the following proposed approaches:

- ***Local Hot***: the IBM's JVM instructions for laying out fields inside objects in the same groups are followed.
- ***General Hot***: memory efficiency is neglected and fields are reordered inside objects only according to their hotness.
- ***General Packed***: a combination of the above approaches.

Suppose an object has the following fields in the order defined by the programmer: F1 (4 bytes), F2 (8), F3 (4), F4 (8), F5 (8), F6 (4), F7 (8), F8 (8).

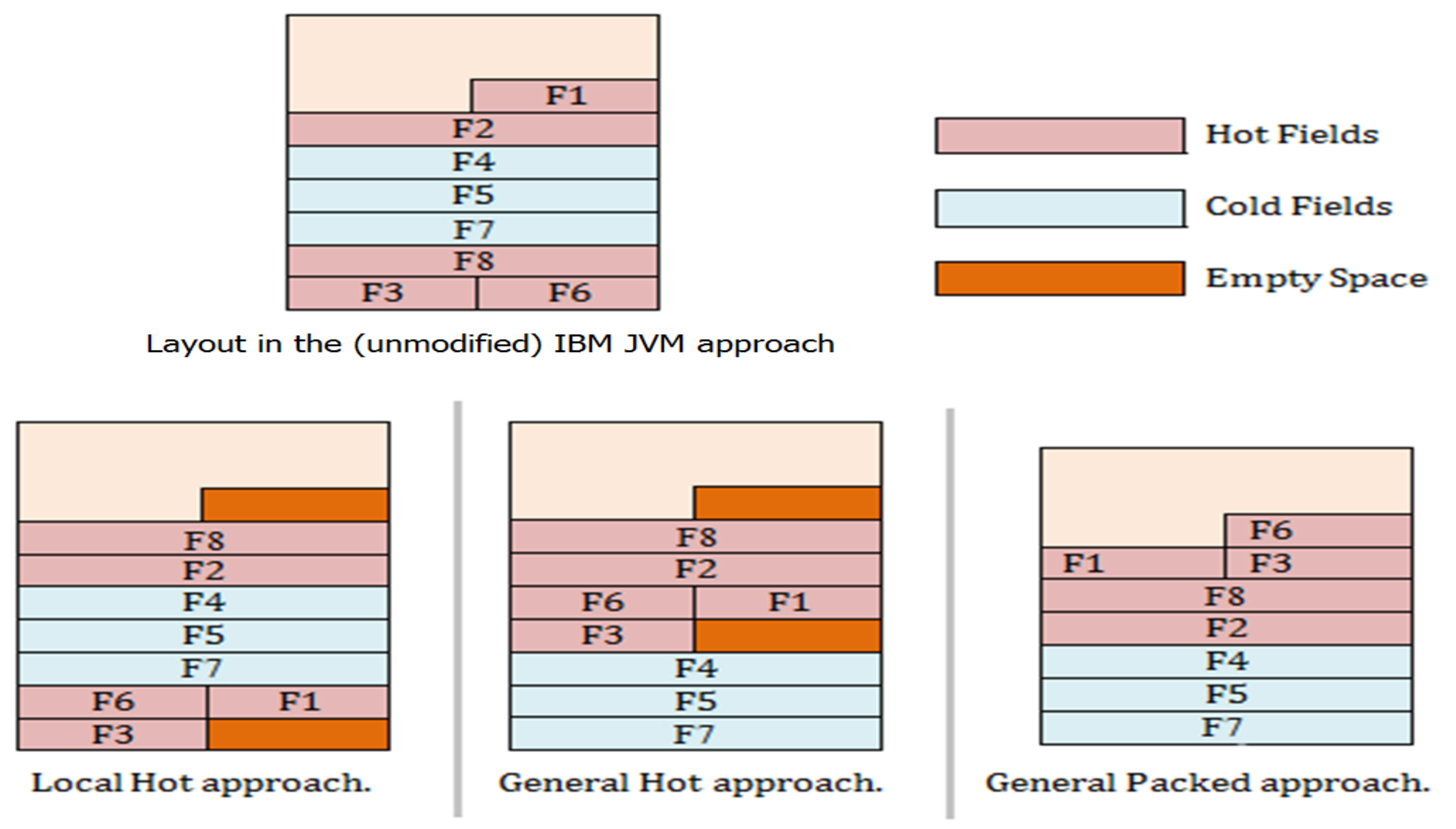
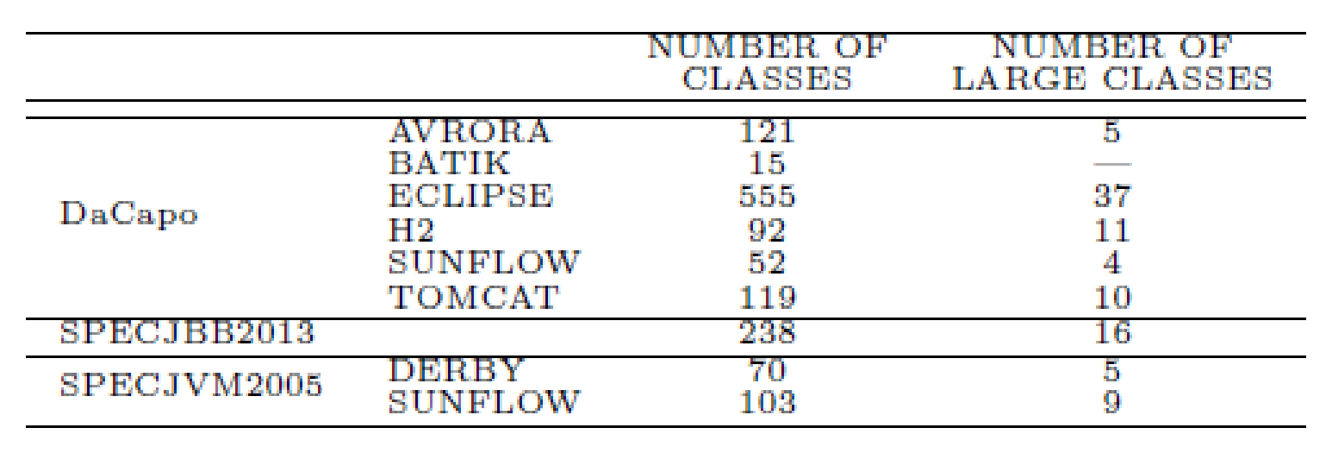


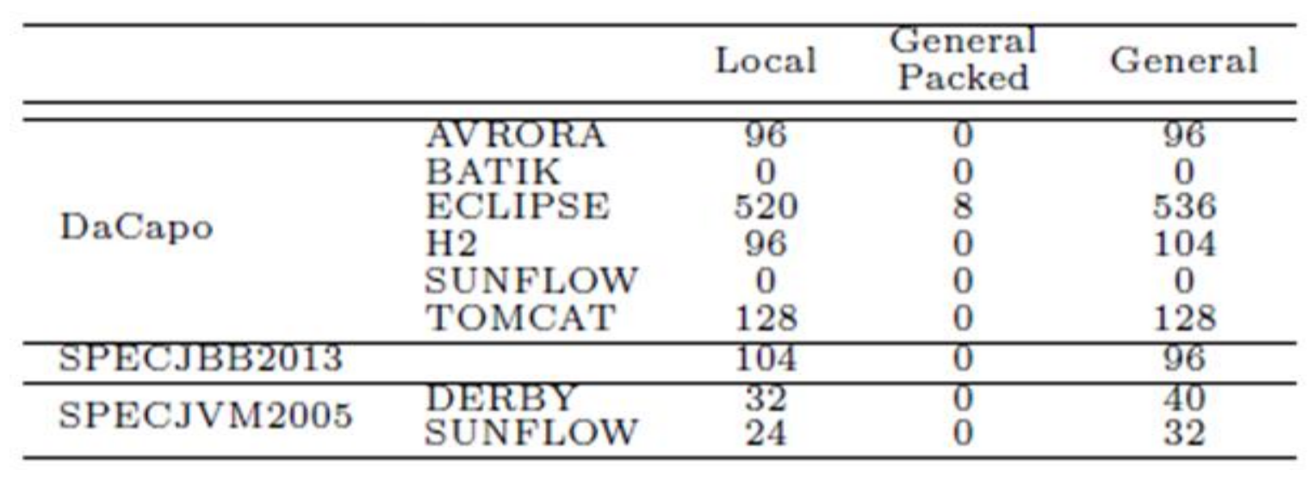
Figure 5: An example of object layout in the different approaches

RESULTS



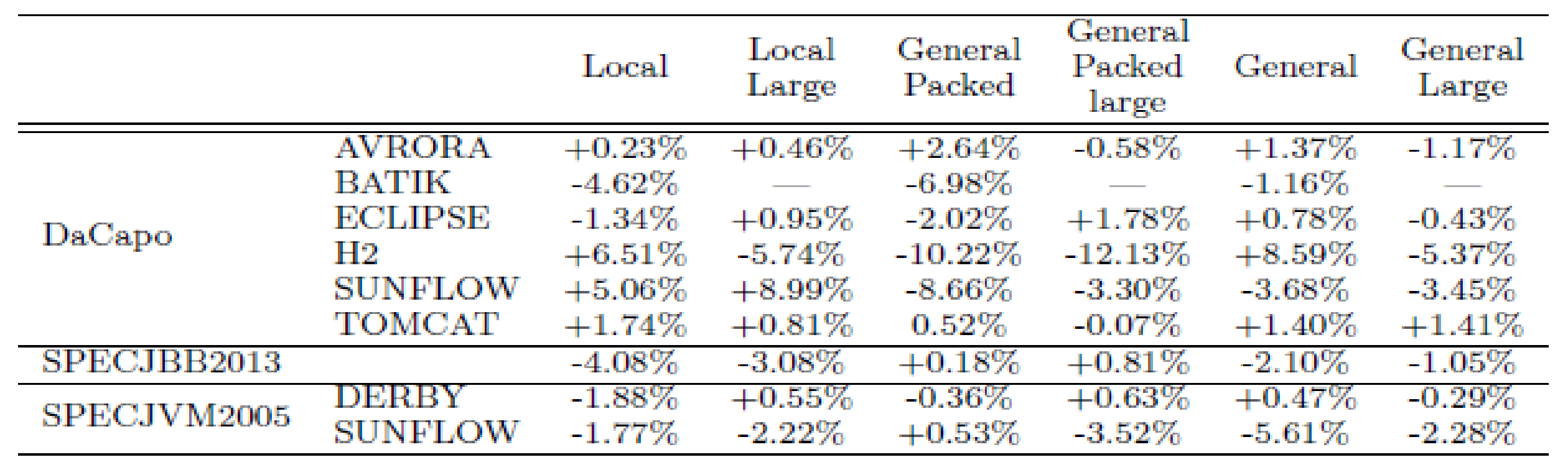
		NUMBER OF CLASSES	NUMBER OF LARGE CLASSES
DaCapo	AVRORA	121	5
	BATIK	15	—
	ECLIPSE	555	37
	H2	92	11
	SUNFLOW	52	4
	TOMCAT	119	10
SPECJBB2013		238	16
SPECJVM2005	DERBY	70	5
	SUNFLOW	103	9

Table 1: Number of classes on which optimization is performed.



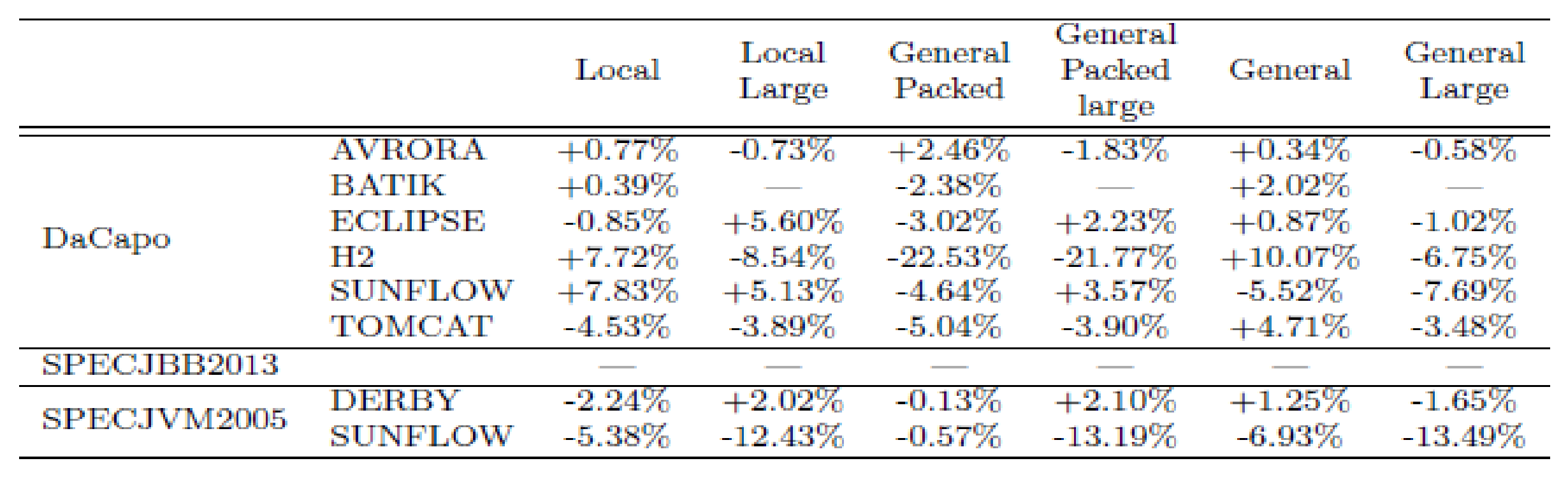
		Local	General Packed	General
DaCapo	AVRORA	96	0	96
	BATIK	0	0	0
	ECLIPSE	520	8	536
	H2	96	0	104
	SUNFLOW	0	0	0
	TOMCAT	128	0	128
SPECJBB2013		104	0	96
SPECJVM2005	DERBY	32	0	40
	SUNFLOW	24	0	32

Table 2: A comparison of the total extra size in bytes added to objects.



		Local	Local Large	General Packed	General Packed large	General	General Large
DaCapo	AVRORA	+0.23%	+0.46%	+2.64%	-0.58%	+1.37%	-1.17%
	BATIK	-4.62%	—	-6.98%	—	-1.16%	—
	ECLIPSE	-1.34%	+0.95%	-2.02%	+1.78%	+0.78%	-0.43%
	H2	+6.51%	-5.74%	-10.22%	-12.13%	+8.59%	-5.37%
	SUNFLOW	+5.06%	+8.99%	-8.66%	-3.30%	-3.68%	-3.45%
	TOMCAT	+1.74%	+0.81%	0.52%	-0.07%	+1.40%	+1.41%
SPECJBB2013		-4.08%	-3.08%	+0.18%	+0.81%	-2.10%	-1.05%
SPECJVM2005	DERBY	-1.88%	+0.55%	-0.36%	+0.63%	+0.47%	-0.29%
	SUNFLOW	-1.77%	-2.22%	+0.53%	-3.52%	-5.61%	-2.28%

Table 3: A comparison of the number of cache misses in different benchmarks.



		Local	Local Large	General Packed	General Packed large	General	General Large
DaCapo	AVRORA	+0.77%	-0.73%	+2.46%	-1.83%	+0.34%	-0.58%
	BATIK	+0.39%	—	-2.38%	—	+2.02%	—
	ECLIPSE	-0.85%	+5.60%	-3.02%	+2.23%	+0.87%	-1.02%
	H2	+7.72%	-8.54%	-22.53%	-21.77%	+10.07%	-6.75%
	SUNFLOW	+7.83%	+5.13%	-4.64%	+3.57%	-5.52%	-7.69%
	TOMCAT	-4.53%	-3.89%	-5.04%	-3.90%	+4.71%	-3.48%
SPECJBB2013		—	—	—	—	—	—
SPECJVM2005	DERBY	-2.24%	+2.02%	-0.13%	+2.10%	+1.25%	-1.65%
	SUNFLOW	-5.38%	-12.43%	-0.57%	-13.19%	-6.93%	-13.49%

Table 4: A comparison of the execution time in different benchmarks.

CONCLUSION

There is almost a 9% increase in the number of cache misses in the worst case and more than a 12% improvement in the best case in different benchmarks. Also, in the worst case, the speed is slowed down by around 10%, and in the best case it is improved by more than 20%.

In the future, field reordering could be done using affinity as well as hotness. In this approach, fields could be reorganized according to the number of times they are accessed together within a short time interval and as a result, the chance that related fields are placed near each other in the cache would be increased.