# Heuristic for profiling bandwidths in object oriented applications

Peter Bertels and Dirk Stroobandt
Ghent University, ELIS
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
peter.bertels@ugent.be

*Abstract*— **The performance of object oriented applications is severely influenced by the access time to the processed data. On a multiprocessor system with distributed shared memory, the average access time can be optimised by allocating the objects in the most appropriate memory. Knowledge of the bandwidths between an object and the different processing elements is important to choose the best allocation. We want to establish dynamic memory allocation on a mixed hardware/software platform. Objects will be moved from one memory to another in order to increase the system performance. Therefore the bandwidth to each object should be measured on the fly, both in hardware and in software. Due to the large number of objects in typical applications and the limited resources, it is not feasible to count data accesses to every object. Only a small number of objects can be tracked. We present a hardware friendly heuristic for dynamically identifying the most accessed objects. Our heuristic also provides estimations of the total number of accesses to each object. Comparison of these estimations with full bandwidth measurements in software shows that the presented heuristic is rather accurate: for some benchmark programs the objects locally responsible for up to 80% of all data accesses are identified.**

*Keywords*— **profiling, bandwidth, hardware-software codesign, heuristic**

## I. Introduction

We are studying a hybrid computing system consisting of a CPU and a Field Programmable Gate Array (FPGA) used to accelerate computationally intensive tasks. This system executes Java applications on a modified Java Virtual Machine (JVM). As section II describes, Java objects can be placed in different memories and access times to those objects differ. Our mixed hardware/software platform allows the objects to be moved from one memory to another. Choosing the optimal location for each object will enhance the system performance.

Knowledge of the bandwidths between an object and the different processing elements, i.e. CPU or methods in hardware, is important to choose the best memory allocation. We like to measure those bandwidths on the fly, both in hardware and in software. Section III shows that this is not trivial, even impossible, due to the large number of objects in typical applications and the limited resources.

We propose to estimate the bandwidths to the most accessed objects instead of measuring the bandwidths to all objects. Section V describes our heuristic algorithm to dynamically identify the most accessed objects. This heuristic algorithm also provides an estimation of the bandwidth to those objects.

This algorithm was implemented and tested (section VI) in a profiler on top of BCel [1]. Results in section VII show that this heuristic performs quite good on the tested benchmark applications.

## II. Hardware/software platform

Our hybrid hardware/software platform is described in [4]. It consists of a general purpose PC and an FPGA board that is plugged into the host computer. The JVM running the application is modified in order to check for each method call whether a hardware implementation for this method is available on the FPGA. If such an implementation is available communication will be delegated to hardware. One of the major advantages of this system is the fact that it is completely transparent to the Java programmer. This transparency allows the use of hardware acceleration without changing the application, nor recompiling it.

The system has a distributed memory: besides the main memory of the host computer, there can be external memory on the FPGA board. All memories are mapped into the same address space. This also is transparent to the programmer. Methods in software and methods in hardware can access the same objects without having to know in which memory they reside. Nevertheless the memory that holds a certain object is very important because access times to this object depend on it.

Communication between the host computer and the FPGA board is very costly [4] and should be avoided. If an object is accessed only by methods in software, the object should reside in main memory on the host computer. Objects which are accessed mostly by methods implemented in hardware should reside in memory on the FPGA board.

Objects can be moved from one memory to another, e.g.

at garbage collection time [3]. In order to move the right objects to the right memory, we need to know how many times objects are accessed by software and by hardware methods.

## III. Profiling bandwidths

We like to measure bandwidths to objects with the intention of establishing dynamic memory allocation on this hybrid hardware/software system. It is important to count accesses to an object from the software and from the hardware separately. Therefore two types of profilers are needed: one in software and one in hardware.

### A. *Profiling in software*

Different approaches are available in order to analyse software execution in general or bandwidth profiling in particular. One approach is instrumentation of the software. Special instrumentation code is inserted in the application. While the program is executed, this code can perform the profiling, e.g. counting data accesses to every object.

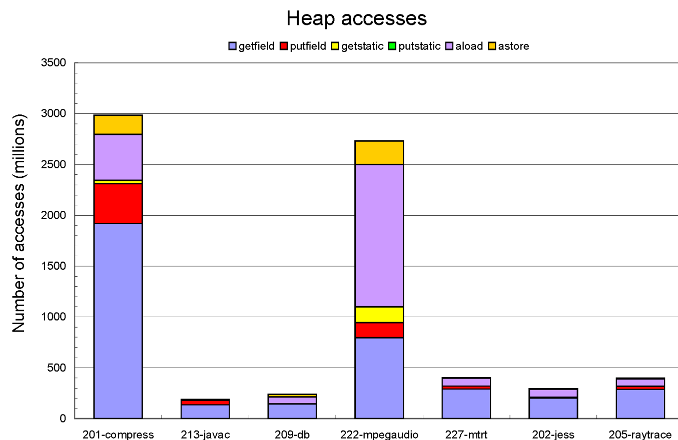


Fig. 1. Total number of heap accesses in several applications of the SPECjvm98 benchmark.

The inserted instrumentation code needs to trap every read and write operation to every object used by the application. Figure 1 shows the total number of read and write operations in several applications of the SPECjvm98 benchmark [2]. Due to the large number of accesses, this full instrumentation introduces a considerable slow down of the execution. Sampled instrumentation can be used to solve this problem. This reduces the overhead but it also decreases the accuracy of the results. In Section VII the influence of sampling on the performance of our heuristic algorithm is studied.

### B. *Profiling in hardware*

On the hardware side a special profiler block can do the profiling in parallel with the functional hardware. This approach causes no overhead in execution time.

There is however another overhead: for each Java object a counter register should be available on the hardware. Due to the large number of objects in typical applications (e.g. several millions in SPECjvm98 programs `_227_mtrt` and `_205_raytrace`), these registers consume a lot of the available resources on the FPGA.

To reduce this area overhead, we have to limit the number of counters. This can be done by identifying the 'most important objects' and only instantiating counters for these objects. Section IV tries to find a good definition for 'important objects' in our use case.

## IV. Definition of 'important' objects

As explained earlier, the end goal of the profiling is to identify objects that are candidates to be moved from memory close to the software side to memory close to the hardware side. It is obviously not useful to move objects which are not accessed frequently.

Moving an object from one memory to another involves copying its data and updating references to this data in the virtual machine and on the hardware platform. This clearly is a costly operation. For frequently accessed objects this can lead to a performance gain, but for less accessed objects the cost for moving them would dominate every possible gain.

The conclusion is that only frequently accessed objects must be considered as important and only accesses to these important objects should be counted. The next section explains our heuristic approach for identifying these important objects on the fly.

## V. Heuristic

Our heuristic algorithm maintains a list of $N$ currently important objects and a counter field for those objects. Each time an object is accessed, the algorithm checks if the object is in the list. If that is the case, its counter will be increased. Objects which are not in the list are considered unimportant and accesses to those objects are simply ignored.

After intercepting $V$ accesses to objects, $R$ least accessed objects are removed from the list. They will be replaced by the first $R$ different accessed objects. As long as the program runs, the list of $N$ important objects is updated this way.

Each time $R$ objects are removed, the counters for the $N - R$ objects remaining in the list are multiplied by

$\alpha \in [0,1]$. With $\alpha$ equal to 0, the algorithm looses information on the history of data accesses in the past. When $\alpha$ is chosen equal to 1, no information is lost. But this is not fair to new objects which have not been counted yet. For our experiments, $\alpha$ is 0.5. This choice eases the future hardware implementation. Our experiments show that this, rather arbitrary choice, performs quite good.

Figure 2 shows the Java method performing this algorithm. The method starts finding the object `o` in the list `objects`. If the object is found in the list, its counter is incremented. We want the list to be sorted, starting with the most accessed objects. This sorting is done incrementally. After incrementing a counter the position of the object `o` is reconsidered. If necessary object `objects[i]` will be swapped with `objects[j]`.

If object `o` is not found in the list but there is room to add a new object, i.e. the list contains less than $N$ objects or `objects[i] == null`, the object `o` is added and its counter is initialised to 1.

At the end of the method a counter `countV` is decremented. This counter was initialised to $V$. When `countV` reaches 0, the $R$ least important objects are removed from the list and the counters for the other objects is rescaled with factor $\alpha$. Afterwards `countV` is initialised to $V$ again.

## VI. Implementation and Tests

We implemented a software profiler in Java. We used the Byte Code Engineering Library (BCel) [1] to manipulate the Java class files of the applications. Before the JVM can execute a Java class, this class has to be loaded with a Classloader. We adapted the BCel class loader in order to instrument the class before the actual execution.

Every read or write operation on a field of a Java object was intercepted. Just before the bytecodes `getfield`, `putfield`, `getstatic`, `putstatic`, `aload` and `astore` a call to our algorithm (`access(Object o)`) was inserted.

In order to test our heuristic algorithm, we divided the program execution in several blocks. Each block contains the same number of read and write operations. During the execution of a block, every read and write operation is intercepted. Whenever an object is accessed, the test function updates one of two counters: one counter for accesses to important objects and another for other object accesses. At the end of an execution block, the value of these counters is exported in a file and the counters are reset for the start of the next block.

The relative number of accesses to 'important' objects gives an indication of the real importance of these objects.

```
public void access(Object o) {
  for (int i = 0; i < N; i++) {
    if (objects[i] == o) {
      counters[i]++;
      if (i != 0) {
        if (counters[i] > counters[i-1]) {
          int j = i;
          while (j > 0 &&
            counters[i] > counters[j-1]) j--;
          int temp1 = counters[j];
          counters[j] = counters[i];
          counters[i] = temp1;
          Object temp2 = objects[j];
          objects[j] = objects[i];
          objects[i] = temp2;
        }
      }
      break;
    }
    if (objects[i] == null) {
      objects[i] = o;
      counters[i] = 1;
      break;
    }
  }
  countV--;
  if (countV == 0) {
    for (int i = 0; i < N - R; i++) {
      counters[i] = (int) Math.round(
              (double) counters[i]* alpha);
    }
    for (int i = N - R; i < N; i++) {
      objects[i] = null;
      counters[i] = 0;
    }
    countV = V;
  }
}
```

Fig. 2. Java code for heuristic algorithm.

## VII. Results

The heuristic algorithm is tested on applications from the SPECjvm98 benchmark. Several combinations for the parameters $N$, $R$ and $V$ were exhaustively tested. For parameter $N$ the following values were tested: 5, 10, 15, 20 and 25. $R$ was in the range $1, \ldots, N$ and $V = m \cdot 10^i$, with $i = 1, \ldots, 6$ and $m = 1, 2, 5$.

The results of these measurements show that our heuristic algorithm performs quite good. This can be seen in Figure 3. For `_201_compress`, `_228_jack` and `_209_db` the quality is even more than 80%.

As could be expected, the parameter $N$ has a great influence of the performance of the algorithm. For a larger $N$, the estimation of important objects is better. This in-

fluence can be seen in Figure 4. This graph also shows that for some benchmarks a relative small list of objects has to be tracked. This is very important for the hardware implementation, where resources are limited.

Some benchmark applications, e.g. `_202_jess`, have a fazed behaviour. At the start of a new faze, when new objects are referenced, the quality of our heuristic drops. But after a while our algorithm 'learns' the new important objects and the quality rises again. This can be seen on Figure 5.

## VIII. Conclusions and Future Work

We presented an algorithm estimate a list of the most accessed objects in an objected oriented environment. Although this research is still in a preliminary stage, the first results are promising. For some of the evaluated benchmark programs our heuristic can effectively estimate the most important, i.e. most frequently accessed, objects.

FUture work includes further enhancement of the algorithm and implementation in hardware and software in order to test the capabilities of this algoritm to establish dynamic memory allocation on a hybrid hardware/software platform.


## Acknowledgements

This research has been supported by a PhD grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).


## References


[1] BCel, the Byte Code Engineering Library. `http://jakarta.apache.org/bcel/`.

[2] JVM Client98 Suite. `http://www.spec.org/jvm98/jvm98/`.

[3] Philippe Faes, Mark Christiaens, Dries Buytaert, and Dirk Stroobandt. FPGA-Aware Garbage Collection in Java. In T. Rissa, S. Wilton, and P. Leong, editors, *2005 International Conference on Field Programmable Logic and Applications (FPL)*, pages 675–680, Tampere, Finland, 1 2005. IEEE.

[4] Philippe Faes, Mark Christiaens, and Dirk Stroobandt. Transparent Communication between Java and Reconfigurable Hardware. In Teofilo Gonzalez, editor, *Proceedings of the 16th IASTED International Conference Parallel and Distributed Computing and Systems*, pages 380–385, Cambridge, MA, USA, 11 2004. ACTA Press.


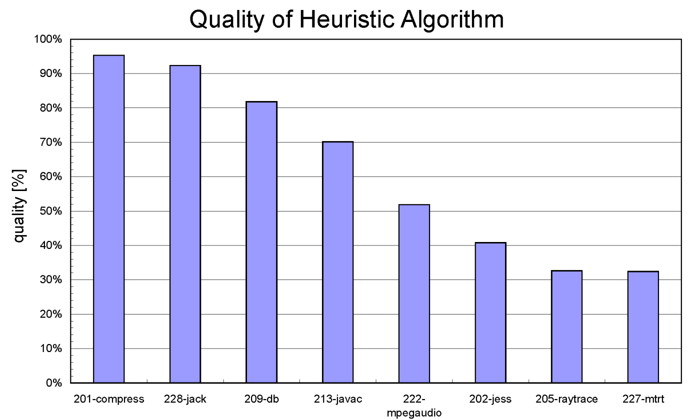


Fig. 3. Average quality for several benchmarks.

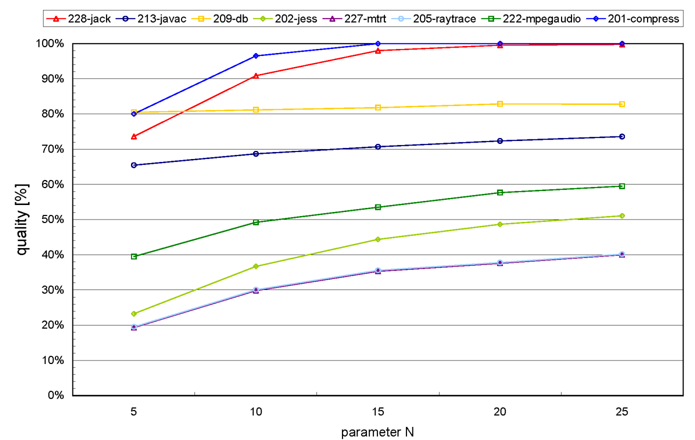


Fig. 4. Quality obtained by the algorithm for several benchmarks and in function of parameter $N$.

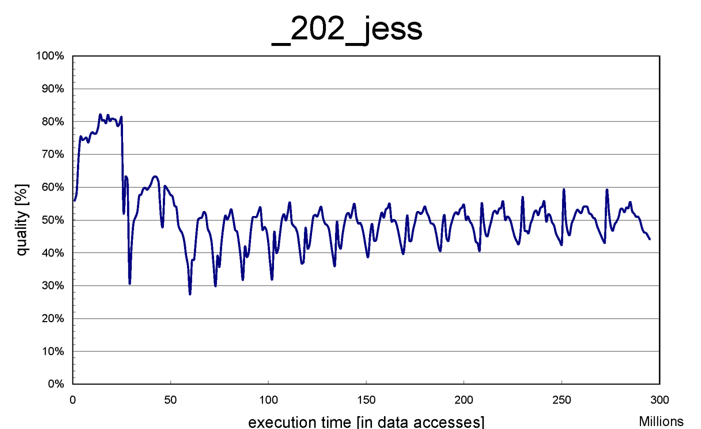


Fig. 5. Quality of the 'important objects' during the execution of the `_201_jess` application. Execution time is expressed in number of data accesses.