Mobility of Data in Distributed Hybrid Computing Systems

Philippe Faes, Mark Christiaens*, and Dirk Stroobandt
Ghent University,
Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium

Abstract

In distributed hybrid computing systems, traditional sequential processors are loosely coupled with reconfigurable hardware for optimal performance. This loose coupling proves to be a communication challenge; the processor units cannot efficiently share a physical memory. This paper proposes a distributed shared memory architecture and a method for effective data migration within that shared memory. Data is moved using a novel garbage collection scheme, the dual semispace *collector. The new garbage collector and the distributed memory prove to be an effective means of data migration in distributed hybrid computing systems.*

1 Introduction

The proliferation of Field Programmable Gate Arrays (FPGAs) as reconfigurable processor units has enabled system architects to combine the computational power of loosely coupled Instruction Set Processors (ISPs) and FPGAs for applications with high computational demands. We call these systems *distributed hybrid computing systems*. Because of their loose coupling, the processor units cannot share one physical memory lest the communication overhead becomes a major bottleneck.

Instead of copying data explicitly between the processor units, we offer the illusion of a shared heap. In reality this heap is distributed over memory chips closely connected to the ISP and FPGA respectively. This paper presents a novel garbage collector algorithm, *dual semispace*, which can move objects within the heap from one physical memory to another. This operation is completely transparent for the programmer.

Unlike page-based distributed memory systems, the garbage collector moves data per object. Objects form a semantic group of data, unlike memory pages, which form fixed-size groups of data. Our results show that the dual semispace garbage collector is an effective alternative for moving data in distributed hybrid computing systems. The dual semispace garbage collector is the first garbage collector, to the best of our knowledge, to move data transparently from one physical memory to another. It relieves the application programmers from dealing with data placement, encapsulating this responsibility in the runtime environment.

The rest of this paper is organized as follows. The next section introduces the problem of data mobility in hybrid distributed computing systems. Section 3 explains how the garbage collector is used for moving data inside a distributed computing system, and introduces the dual semispace garbage collector. Section 4 outlines a demonstration set-up of our fully functional system, and presents timing results for a real-life application. Related work is outlined in Section 5. The paper concludes and looks forward to future work in Section 6.

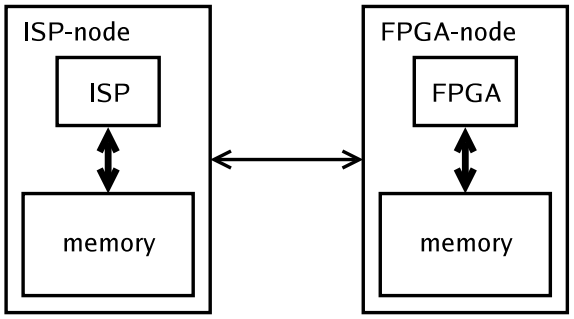

Figure 1. Physical interconnection of the ISP-node and the FPGA-node

2 Data placement in distributed hybrid computing

Our hybrid distributed computing system, as shown in Fig. 1, has two cooperating processor nodes: an ISP node and an FPGA node, connected through a relatively slow

*Mark Christiaens is currently with Symantec Corporation, Antwerpsesteenweg 19, B-9080 Lochristi, Belgium

1-4244-0910-1/07/$20.00

link. Each node has its own memory, with a high-speed connection to its processor unit; the memory in the ISP node is tightly connected to the ISP itself, and likewise for the FPGA node.

The ISP runs a computationally intensive Java application, and uses the FPGA for the acceleration of some of the Java methods. Due to the specific computation model of FPGAs, their massively parallel computation power and their relatively low clock frequency, not all methods are candidates for FPGA acceleration. Candidate methods have a high degree of low-level parallelism and little sequential control-dominated code. These methods can be found in image processing, bio-informatics applications, scientific computing, etc.

Whenever an FPGA implementation is available for a candidate method, the Java runtime may choose to intercept the software execution of that method and let it run on the FPGA instead. The intercepted Java thread will stall until the FPGA computation is finished, while other Java threads can continue their work.

2.1 Data transfer

Now how can we pass the data needed for a computation to the FPGA and back? Two options arise. First, the application could explicitly write all data elements to memories on the FPGA node and read results back from these memories. This has the disadvantage that the communication is explicitly visible in the software, and that it leaves no room for lazy copying. The second option is to copy only a *reference* to data-structures (Java objects), and let the FPGA fetch the data whenever it is needed. The latter option has two problems, both of which find their solution in adapting the garbage collector.

As a first problem, the Java garbage collector moves objects around in order to avoid memory fragmentation. If the FPGA uses a pointer to an object that has been moved, the system will inevitably crash. We have discussed and solved this problem in prior work [1], by adapting the garbage collector. The adapted collector knows about the references from the FPGA into the Java heap and updates these references during the compaction phase.

The second problem is data bandwidth. Each time the same object is used in the FPGA, the same data is copied over the slow FPGA-ISP connection. One could consider keeping a copy of the object in the FPGA for later reuse, effectively *caching* the data. But what if the ISP changes the data in between FPGA invocations? The two versions of the object would be inconsistent. Moreover, the traditional coherence and invalidation techniques [2, 3] may prove difficult or impossible to implement over an arbitrary FPGA-ISP connection. The solution to this problem, a distributed shared heap, is the subject of this paper.

2.2 Distributed shared heap

In order to alleviate the bandwidth problem, we choose to allocate part of the Java heap on the FPGA board. This way, the Java runtime environment (more precisely, the garbage collector) can place data from either on the ISP node or on the FPGA node, potentially bringing the it closer to its consumer. The placement of objects can be driven either by *heuristics* or by *programmed hints* to the JVM.

Both processor units have access to both parts of the heap as depicted in Fig. 2. When an object is moved to the FPGA node, the FPGA has fast access to it. The ISP can still address the object transparently, but at a lower bandwidth.

The FPGA makes its memory visible on the system bus. From there, the operating system maps the FPGA memory into the memory space of the JVM process, enabling the JVM to use this extra memory as heap memory. Only minimal modifications were required to the memory management subsystem for setting up the FPGA memory.

Unlike modern ISPs, FPGAs do not have a Memory Management Unit (MMU). We have built a minimal MMU to provide transparent access from the FPGA to the main memory and the FPGA memory. From the FPGA side, the MMU behaves like a memory, with varying response times. On the other side, the MMU has read/write access to the on-board DDR RAM, and can request data transfers from the JVM. When the FPGA requests a memory transaction from the MMU, the action of the MMU depends on the address of the transaction. If this address is within the range of the on-board DDR RAM, the MMU can directly access the DDR RAM. If the address is in the range of the ISP's main memory, the MMU forwards the request to a software thread in the JVM.

In future versions, the MMU will handle accesses to main memory by performing address translation and then fetching the data *directly* from main memory. On systems with virtual memory, address translation will require communication with the operating system's page tables.

Independent of the action taken by the MMU (reading from local memory, communicating with the JVM or fetching directly from main memory) the behavior observed from the FPGA point-of-view is identical except of course for response times; the protocol only requests a memory read or write operation.

In effect, our system uses a single distributed shared memory, the Java heap, which has Non-Uniform Memory Access times (NUMA) from the different computation units. The next section discusses the memory management of the NUMA heap.

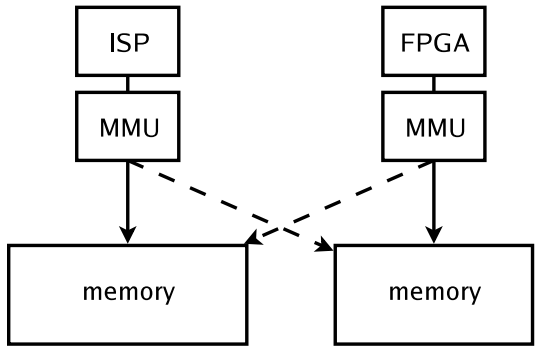

Figure 2. Using memory management units to abstract the physical location of data from the processing units

Algorithm 1 Semispace garbage collection algorithm

```
swap(fromSpace,toSpace)
todoSet = emptySet
for object in rootList:
  todoSet.add(object)
//scan all reachable objects
while !todoSet.empty():
  object = todoSet.pop()
  newObject = object.copyTo(toSpace)
  object.link = newObject
  for child in newObject.references:
    if child.isIn(fromSpace):
      todoSet.add(child)
// update references in objects
for object in toSpace:
  for child in object.references:
    if child.isIn(fromSpace):
      object.updateRef(child,child.link)
```

3 Managing the NUMA heap

The previous section describes our rationale for using a NUMA heap in our hybrid distributed computing system. Because of its specific characteristics —references from the FPGA to heap objects and the heap's distribution over non uniform memory chips— a specific memory management strategy is required. For this purpose, we have implemented a new garbage collector scheme in the Memory Management Toolkit (MMTk) [4], which is used in the Jikes Research Virtual Machine (JikesRVM, formerly Jalapeño [5]).

3.1 Garbage Collection

A dynamic memory management system, or Garbage Collector (GC), is a component of a runtime environment charged with removing unused objects. It is automatically triggered whenever the memory is exhausted in order to create new free memory. The GC selects an object for removal when it is no longer addressable by the application, i.e. when the application no longer has a (direct or indirect) reference to that object. In this case the object is called *dead*, and it may safely be removed by the garbage collector.

In order to determine which objects are dead, the GC explores the object-reference graph, a graph where each node represents a Java object and each directed edge represents a reference from one object to another. In this graph, some nodes are designated as *root nodes*. They represent objects that are always reachable, such as the stacks of each thread, and static references. From these root objects, the connectivity problem is solved. All objects that are connected to a root node are alive; the others are dead. The memory of dead objects is freed and becomes available for new allocations.

Many GC algorithms also include *compaction* in order to prevent memory fragmentation. The memory compaction places all living objects side to side, leaving one contiguous memory area free for future allocations. The next subsection discusses the semispace garbage collector, an algorithm that performs collection and compaction at the same time.

3.2 Semispace

The semispace garbage collector [6], presented as Algorithm 1, divides the heap in two equal parts, labeled *to-space* and *from-space*. Whenever the GC is not active, only the *to-space* is used and all new objects are allocated in the *to-space*.

When a garbage collection is triggered, the *to-space* becomes the new *from-space* and vice-versa. The root-objects, and objects that are (directly or indirectly) referenced by root-objects are recursively copied to the new to-space. The new object address is written in the old object header. Using this header, all references to the old objects are updated to refer to the newly copied objects. Unreachable objects are not copied but stay in the *from-space*, which is discarded. New objects are again allocated in the *to-space*.

3.3 Multiple spaces

While the semispace collector requires double the amount of memory as is actually used by the objects, it is a simple and straightforward algorithm, and many collection schemes are based on the semispace collector. Many collectors use different spaces for different kind of objects.

Generational collectors place new objects in a *nursery-space*, and copy older objects to a *mature-space*. The *nursery-space* is collected often, and because most objects die young, relatively few objects ever get copied to the

Algorithm 2 Dual Semispace garbage collection algorithm

```
swap(ispFromSpace,ispToSpace)
swap(fpgaFromSpace,fpgaToSpace)
todoSet = emptySet
for object in rootList:
  todoSet.add(object)
//scan all reachable objects
while !todoSet.empty():
  object = todoSet.pop()
  if( (requestedSpace(object) == ISP_SPACE
      and ispToSpace.hasRoomFor(object))
      or !fpgaToSpace.hasRoomFor(object)):
    newObject = object.copyTo(ispToSpace)
  else:
    newObject = object.copyTo(fpgaToSpace)
  object.link = newObject
  for child in newObject.references:
    if (child.isIn(ispFromSpace)
       or child.isIn(fpgaFromSpace)):
      todoSet.add(child)
// update references in objects
for object in (ispToSpace + fpgaToSpace):
  for child in object.references:
    if (child.isIn(ispFromSpace)
       or child.isIn(fpgaFromSpace)):
      object.updateRef(child,
                       child.link)
```

mature-space. The *mature-space* is only collected sporadically. Also, some collectors place all objects whose size exceeds a given number in the *large-object-space*. The fact that this space contains few large objects makes it more efficient to collect it.

Each space contains objects with specific characteristics. The *to-space* contains live (or recently deceased) objects while the *from-space* contains only void. The *nursery-space* contains young objects, the *mature-space* contains old objects and the *large-object-space* contains large objects. In our dual semispace, we introduce a new pair of heap spaces, which contain objects that show affinity to the FPGA.

3.4 Dual semispace

In the dual semispace strategy, each physical memory (one connected to the ISP and one connected to the FPGA) is separated into two spaces. The algorithm, presented as Algorithm 2 is similar to the semispace garbage collector, but now live objects can be copied to either of the *to-spaces (isp-to-space* and *fpga-to-space*). A heuristic determines which of both spaces is preferred. If there is enough room left in the selected space, the object is moved there. If not, the object is moved to the other *to-space*. Note that both *to-spaces* are full only when the system runs out of heap-space entirely.

The dual semispace algorithm is only measurably slower than the semispace algorithm to the extent that data is copied over a slower memory connection. Also, the extra data structures needed for the dual semispace are constant in size.

While it is possible to extend dual semispace to n-semispace for systems of multiple ISPs and FPGAs, this extension is not trivial. Since many smaller semispaces are used, it is very likely that the size of a semispace will be a limiting factor for placing objects. It is not clear which behavior is preferred when the heuristic in use wants to place an object in an already full to-spaces. This question is subject to future research.

3.5 Heuristics

Since only a few time-consuming methods are executed on the FPGA, most Java objects are never accessed by the FPGA. Only objects that *are* accessed by the FPGA are candidates for transferal to the FPGA-memory. Because of this asymmetry, the data placement algorithm in the garbage collector can be simplified.

The current implementation is not a heuristic, but is based on programmed hints. It keeps a list of (weak) references to objects that should preferably be placed in the FPGA-memory. All other objects are placed in the ISP-memory.

In the future we will investigate heuristics. One possible heuristic is to move all objects for which the FPGA holds a reference. This is easy to implement, since the garbage collector has access to all references from the FPGA to the heap. However, some references may not be used at all or very infrequently by the FPGA. Having the GC copy the entire object to the FPGA memory may be too expensive.

The execution could also be instrumented so that accesses to each object are logged, perhaps in a sampled manner. An object is then moved to the FPGA memory iff

$$\begin{aligned} & A_{ISP}T_{ISP,FPGA}+A_{FPGA}T_{FPGA,FPGA} \\ > \; & A_{ISP}T_{ISP,ISP}+A_{FPGA}T_{FPGA,ISP} \end{aligned}$$

where A_X is the number of accesses to the object from processor unit X, and $T_{X,Y}$ is the time needed for one access from processor unit X to memory Y. This strategy assumes that the behavior of the application is static, i.e. that the cost of moving data will eventually be compensated by the time gained from the more efficient data placement. In applications with strong dynamic behavior, heuristics will have to make non-trivial predictions of the future, a problem akin to that of cache strategies and branch prediction.

In the next section we demonstrate the speedup from data migration using the dual semispace garbage collector.

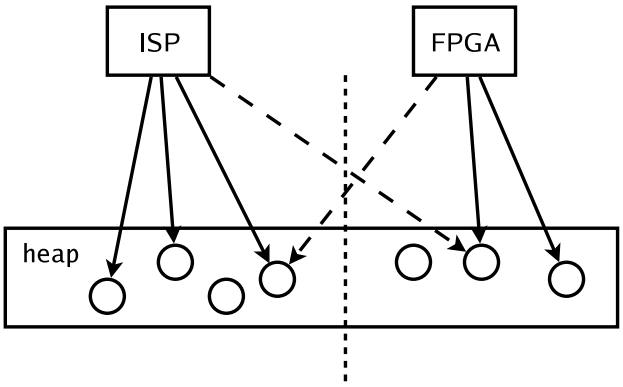

Figure 3. Logical NUMA heap

4 Mobility demonstrated

As described earlier, both data and computational power can be moved at run-time. The data is moved by the dual semispace garbage collector and the computation is either on the ISP, or it is intercepted by the JVM and sent to the FPGA.

This section describes how data and computation are moved in a protein sequence aligner application [7]. The application performs a comparison of protein sequences using the Smith-Waterman [8] dynamic programming algorithm.

The system consists of a desktop computer with an Intel Pentium 4 processor at 2.4 GHz, which is connected through a PCI link to an Altera PCI Development board, carrying a Stratix 1s25c5 FPGA and 256 MiB of DDR SDRAM. Memory accesses from the ISP to its own memory are as fast as 1246 MiB/s, The bandwidth over the FPGA-ISP is limited to 73 MiB/s, and the bandwidth of the FPGA to its own memory is 630 MiB/s.

The application has one *hot* method (`align`) that is executed often and consumes most of the computation time. The method also makes many accesses to the same data structure, the *substitution matrix*. We have implemented this method on the FPGA, making it possible to choose on which processing unit it will be executed. The experiment comprises four phases, A through D, as depicted in Table 1.

In phase A all data and computation are on the ISP node. In phase B, the hot method is moved to FPGA. Next, the array is moved to the FPGA memory in phase C. Finally in phase D, computation is moved back to the ISP. Fig. 4 shows the number of executions of `align` per second.

At first, the performance of phase A is low because many classes need to be loaded into the virtual machine. As time progresses, the JikesRVM recompiles the time-critical methods in an optimized way. During the recompilation the performance is lower, only to yield a better performance when recompilation finishes.

phase	A	B	C	D
computation	ISP	FPGA	FPGA	ISP
data	ISP	ISP	FPGA	FPGA

Table 1. Four phases of data mobility

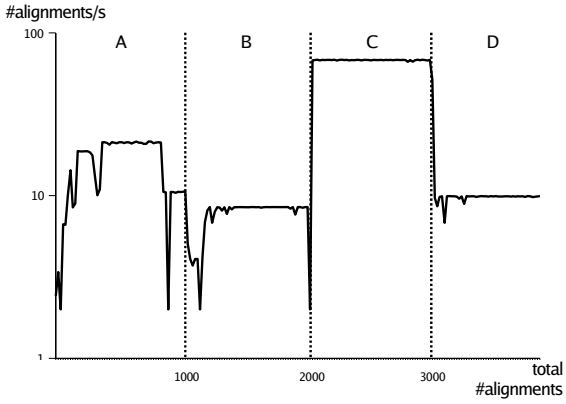

Figure 4. Number of executions per second of the `align` method in different execution phases

Since many new Java classes are needed for the communication with the FPGA, the same pattern of gradually increasing performance can be seen in the beginning of phase B. Even though `align` can be computed faster on the FPGA than on the ISP, the performance in phase B is lower than in phase A, due to the communication overhead. The application can only benefit from the FPGA's computation power once the substitution matrix is moved to the FPGA memory, in phase C. When the computation is again moved to the ISP, in phase D, the performance decreases again. The two better performing phases A and C are, not surprisingly, the phases in which data and computation are on the same node.

In this example, the computation and the data were moved dynamically by giving hints to the JikesRVM. In more complex situations, where many methods and many data-structures are mobile, the optimal solution may not be intuitively obvious. However, it will still be easy to explore the design space using the dual semispace garbage collector and the adapted JikesRVM.

5 Related work

We have investigated the problem of data mobility in a distributed hybrid computing system. This is akin to the problem of shared memories in Non Uniform Memory Access (NUMA) and Cache Coherent NUMA (cc-NUMA) systems [9, 10, 11]. In these systems memory pages are

used as elementary data unit, and are synchronized between computation nodes. A disadvantage of page-based data mobility is the possibility of *false data sharing*, in which two unrelated data items are allocated on the same memory page. This memory page then needs be sent back-and-forth between two computation nodes who use those data items.

A first attempt at using the garbage collector in distributed computing was done by Tikir et al. [12]. This work focuses on traditional cc-NUMA machines, with all processors being ISPs. Also, the paper presents a simulated projection based on their proposed heap lay-out, without presenting a GC algorithm.

Existing hybrid computing systems either have one shared memory between the FPGA and ISP [13, 14], or use message-based communication [15].

6 Conclusion and Future work

This paper presents a novel approach to data mobility in hybrid distributed computing systems. To the best of our knowledge no prior work presents a working garbage collector used for data mobility in a NUMA-heap.

The garbage collector provides a transparent way of moving data closer to a given processor unit. While neither processor needs to worry about the exact data location, the runtime environment places data where it best sees fit.

In future work, we will implement heuristics to drive the dual semispace collector. We will also investigate other garbage collection schemes and their interaction with heuristics.

Acknowledgments

This research in part is supported by grant 020174 of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen) and by hardware and software donations from Altera. Philippe Faes is supported by a Ph.D. grant of the IWT-Vlaanderen.

References

[1] Ph. Faes, M. Christiaens, D. Buytaert, and D. Stroobandt, "FPGA-aware garbage collection in Java," in *2005 International Conference on Field Programmable Logic and Applications (FPL)* (T. Rissa, S. Wilton, and P. Leong, eds.), (Tampere, Finland), pp. 675–680, IEEE, 1 2005.

[2] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," pp. 280–289, 1988.

[3] W. Weber and A. Gupta, "Analysis of cache invalidation patterns in multiprocessors," *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pp. 243–256, 1989.

[4] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Oil and water? High performance garbage collection in Java with MMTk," in *ICSE 2004, 26th International Conference on Software Engineering Edinburgh, Scotland, United Kingdom*, pp. 137–146, 5 2004.

[5] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley, "The Jalapeño virtual machine," *IBM Systems Journal*, vol. 39, no. 1, pp. 211–238, 2000.

[6] R. Fenichel and J. Yochelson, "A LISP garbage-collector for virtual-memory computer systems," *Communications of the ACM*, vol. 12, no. 11, pp. 611–612, 1969.

[7] Ph. Faes, B. Minnaert, M. Christiaens, E. Bonnet, Y. Saeys, D. Stroobandt, and Y. Van de Peer, "Scalable hardware accelerator for comparing dna and protein sequences," in *Proceedings of the First International Conference on Scalable Information Systems*, (Hong Kong), 5 2006.

[8] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[9] B. Falsafi and D. Wood, "Reactive NUMA: A Design For Unifying S-COMA And CC-NUMA," *Computer Architecture, 1997. Conference Proceedings. The 24th Annual International Symposium on*, pp. 229–240, 1997.

[10] A. Cox and R. Fowler, "The implementation of a coherent memory abstraction on a NUMA multiprocessor: experiences with platinum," *Proceedings of the twelfth ACM symposium on Operating systems principles*, pp. 32–44, 1989.

[11] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," in *Proceedings of the fifth annual ACM symposium on principles of distributed computing*, nov 1986.

[12] M. Tikir and J. Hollingsworth, "NUMA-Aware Java Heaps for Server Applications," *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pp. 108b–108b, 2005.

[13] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The Molen polymorphic processor," *IEEE Transactions on Computers*, pp. 1363–1375, November 2004.

[14] E. Lattanzi, A. Gayasen, M. Kandemir, V. Narayanan, L. Benini, and A. Bogliolo, "Improving Java performance by dynamic method migration on FPGAs," in *Proceedings of RAW 2004*, 2004.

[15] Y. Ha, G. Vanmeerbeeck, P. Schaumont, S. Vernalde, M. Engels, and H. De Man, "Virtual Java/FPGA interface for networked reconfiguration," pp. 558–563, Jan. 2001.