Strategies for Dynamic Memory Allocation in Hybrid Architectures

Peter Bertels
peter.bertels@ugent.be

Wim Heirman
wim.heirman@ugent.be

Dirk Stroobandt
dirk.stroobandt@ugent.be

Department of Electronics and Information Systems
Ghent University, Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium

ABSTRACT

Hybrid architectures combining the strengths of general-purpose processors with application-specific hardware accelerators can lead to a significant performance improvement. Our hybrid architecture uses a Java Virtual Machine as an abstraction layer to hide the complexity of the hardware/software interface between processor and accelerator from the programmer. The data communication between the accelerator and the processor often incurs a significant cost, which sometimes annihilates the original speedup obtained by the accelerator. This article shows how we minimise this communication cost by dynamically chosing an optimal data layout in the Java heap memory which is distributed over both the accelerator and the processor memory. The proposed self-learning memory allocation strategy finds the optimal location for each Java object's data by means of runtime profiling. The communication cost is effectively reduced by up to 86% for the benchmarks in the DaCapo suite (51% on average).

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*Shared memory*; D.3.4 [**Programming Languages**]: Processors—*Memory management*; D.4.2 [**Operating Systems**]: Storage Management—*Distributed memories*

General Terms

Algorithms, Experimentation, Performance

Keywords

Hardware acceleration, Java, Memory management

1. INTRODUCTION

Hardware accelerators or other application-specific coprocessors are used to improve the performance of computationally intensive programs. Large speedups are achieved by exploiting massive parallelism in hot code segments [3]. Recently, the same methodology has been applied to Java programs [4]. Two main directions can be identified in this domain: acceleration of the Java Virtual Machine (JVM) itself and acceleration of specific methods of Java programs. In the first approach additional hardware provides specific Java functionality such as thread scheduling and garbage collection or the translation of bytecode to native instructions [9]. A further evolution of this idea is a Java-specific processor that natively executes bytecode [10]. In the second approach, an application-specific hardware accelerator executes in parallel with the main processor. Additionally, we consider the hardware as an integral part of the JVM rather than an application-software controlled device. An advantage of this approach is that the hardware acceleration is transparent to the Java programmer. If the hardware accelerator is reconfigurable, functionality can even be moved dynamically from the general-purpose processor to the accelerator [5]. Hardware execution is then an additional optimisation step in the just-in-time compiler, where the hardware configuration can be loaded from a library [2] or even generated on-the-fly [6].

In this article we concentrate on the latter approach because it uses the hardware only for specific functions where a significant speedup can be obtained as has been shown in several hardware implementations [3]. Less hardware-friendly methods are left in software which contrasts to the first approach where often a significant amount of hardware resources needs to be allocated for functionality like memory management, scheduling . . . Our approach leads to a hardware accelerated JVM which is described in Section 2. In such a hybrid architecture, the JVM has to manage scheduling and memory allocation, also for functionality executed in hardware. Java's shared-memory model now extends to the accelerator's local memory. The JVM must be extended such that both native Java code and the accelerator can access all objects, independent of their physical location.

An important task of the JVM is the placement of objects in the distributed Java heap. Since the accelerator is usually connected through a relatively slow communication medium, remote memory accesses are costly and should thus be avoided as much as possible. To this end, the JVM should allocate objects in the memory region closest to the most prolific user of the data. This way, data private to a thread is always in local memory, which minimises communication overhead. A static analysis is not sufficient for solving the object placement problem as it can only estimate which data are private to a method conservatively. More-

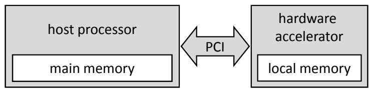

Figure 1: Hybrid hardware platform

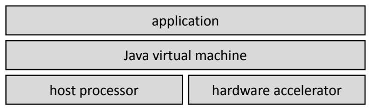

Figure 2: The JVM hides the complexity of the underlying hybrid architecture from the application.

over, shared data can be accessed asymmetrically by the different system components, the ratio in accesses among components is often data dependent and thus hard to estimate at compile-time. Finally, when functionality is dynamically migrated between general-purpose processor and hardware accelerator, a runtime approach can no longer be avoided.

We propose several techniques for communication-aware memory management in Section 3. For each Java object, the optimal memory location is determined based on the usage pattern of this object. The self-learning approach tries to estimate the usage patterns for each object based on measured patterns for previously allocated objects. This technique is compared to a baseline algorithm which does not take communication cost into account, and to a static technique for local memory allocation which tries to reduce communication overhead without actually measuring data access patterns. Our data placement strategies lead to a reduction of the remote memory accesses by up to 86% (51% on average) for the DaCapo benchmarks (Section 4).

2. HARDWARE ACCELERATED JVM

2.1 Host processor and hardware accelerator

In this work we use the classical concept of an accelerator as a coprocessor: the hardware platform is a hybrid architecture, consisting of a general-purpose host processor and an application-specific hardware accelerator (Figure 1). The accelerator executes a small but computationally intensive part of the Java application, while the host processor executes the remainder of the application. In this hybrid architecture, the processor and the accelerator both have their own local memory. The connection between the two components is realised by means of a relatively slow bus such as PCI, HyperTransport, ... Therefore, effective use of this platform is usually limited to algorithms with a high computation to communication ratio. For this class of applications a significant speedup can be obtained by exploiting the massive parallelism available on FPGAs or ASICs [8].

2.2 JVM as hardware abstraction layer

We want to hide the complexity of managing control flow and communication between the accelerator and the host processor from the programmer. Also, when the hardware accelerator is reconfigurable, functionality can be moved dynamically from the host processor to the accelerator by loading the appropriate configuration from a library or even by generating a hardware implementation of the Java code on-the-fly. This is possible when we consider the JVM to be an abstraction for the underlying hardware (Figure 2). The JVM can now intercept method calls for which a hardware equivalent is available, and delegate execution to the appropriate accelerator. It also enables the accelerator to access objects on the Java heap which is distributed between both main memory and the accelerator's local memory.

In this concept, the hardware is an integral part of the JVM but is invisible to the Java application. Therefore we need to properly define an equivalence between the hardware component and software concepts in the Java language. In our approach, hardware accelerators encapsulate the functional behaviour of the bytecode in the corresponding Java method. This accelerator is considered stateless. At each invocation, both the parameters of the function and the corresponding state —the class for static methods, an object reference for virtual methods— need to be transferred to the hardware component.

2.3 Shared-memory model

Our hybrid architecture uses a shared-memory model that allows both the host processor and the accelerator to access all objects. The Java heap is distributed between main memory and the accelerator's local memory. The garbage collector is extended to account for objects and references in both memories. Whether new objects are placed in main memory or in the accelerator's local memory, should depend on their access pattern. This is exactly the focus of our algorithm for communication-aware data placement which is described in Section 3. Although object-oriented languages like Java strongly emphasize the connection between the object's data and its functionality (methods), in our approach the decisions on data and method placement are treated separately. Indeed, a single object class may have some methods implemented on the accelerator while others are executed by the host processor.

Throughout this paper, we assume that, if the processor caches its accesses to main memory, coherence is in some way maintained when the accelerator writes to an object in main memory, for instance using a coherent HyperTransport bus. During our communication measurements, we assumed that no caching of remote accesses is performed; by the accelerator to main memory or by the host processor to the accelerator's local memory. Since FPGAs have no local cache, this is usually the case on our target platform. On implementations that do support remote caching, we overestimate the communication, by an amount proportional to the hit rate of remote accesses in the cache. Still, optimising the object placement will reduce the communication overhead and may reduce cache requirements for remote addresses.

3. STRATEGIES FOR DATA PLACEMENT

Objects should be placed close to the component (host processor or hardware accelerator) that references them the most. This way a large fraction of memory accesses will be local, minimising communication and its associated cost. Finding the optimal placement at runtime is infeasible for two reasons. First, we don't know the future usage pattern of newly created objects, so we have to base our decision on other information such as profiles, previous usage of other objects ... Second, there are too many objects to keep track of these statistics on a per-object basis. Therefore, we take the placement decision clustered per *creation site*. This is the line in the source code where the objects are created.

Objects created at the same creation site are expected to have a similar usage pattern. Therefore, we can allocate them in the same memory, and have a performance close to that of optimally allocating each object individually. Moreover, we can use measured access patterns of previous objects with the same creation site to determine the optimal allocation site for new objects. These previous access patterns can be measured either on-the-fly using runtime instrumentation or during a separate profiling phase. Some software patterns can break the general rule that creation site and usage patterns are closely related. For example, in class factories a single creation site creates objects of different types which can be used in very different contexts. However, as will be evident in Section 4, for the objects causing most of the remote accesses the connection between the creation site and the usage patterns turns out to be strong.

We will compare several algorithms for communication-aware object placement: a baseline algorithm, optimal placement, local allocation and self-learning allocation. These algorithms differ in implementation complexity, and whether the allocation site is adaptive or fixed and whether it is based on runtime or profile information.

Baseline algorithm.
This algorithm allocates all objects in main memory. No account is taken of the hybrid nature of our architecture, and all memory accesses performed by the accelerator will be remote accesses. Therefore the communication cost will be high although for some benchmarks (Section 4) the difference is acceptable. Implementation complexity is very low since essentially no decision has to be made. The runtime overhead of this strategy is zero.

Optimal placement.
Based on the joint usage pattern for all objects with the same creation site and measured during a complete run of an application, the optimal memory per creation site can be determined. Although this strategy is not 'really optimal' because it does not consider each object individually, we consider it as an 'optimal' implementation within the given constraints and use it to compare all the other strategies. The runtime overhead of this strategy is low, although a separate profiling phase is needed which can be inaccurate due to input-dependent behavior.

Local allocation.
Many objects are allocated on the stack or have a very short lifetime. They are therefore often used almost exclusively by the method which created them. This observation leads to the local allocation strategy which allocates all objects in memory closest to the component that creates them. The information needed to implement this strategy is easily available at runtime, implementation of local allocation is thus straightforward and incurs no runtime overhead nor a separate profiling phase.

Self-learning allocation.
In this strategy, the virtual machine decides at runtime where to allocate objects based on the usage patterns of previous objects. This is particularly useful in the dynamic environment of a hardware accelerated JVM, which decides at runtime whether to execute functionality on the general-purpose processor or on specific hardware accelerators. The JVM continuously counts all memory accesses from both the main processor and the accelerators to each object in both memories. This can for instance be done through (sampled) instrumentation or hardware assisted profiling. Each creation site has its own set of counters, one for the processor and one for the accelerator, each aggregating the number of accesses to objects created at this site. At each point in time, comparing the two counters will tell us which system component has accessed these objects the most up to now. New objects created at this creation site will be allocated in the memory closest to the component with the highest number of accesses. At the end of the program the counters will reach the value obtained during the profiling for optimal placement (assuming a constant input set). The behaviour of the self-learning algorithm will therefore converge towards the optimal placement. The rate of convergence is usually very fast, as shown in Section 4, resulting in a near-optimal remote access ratio.

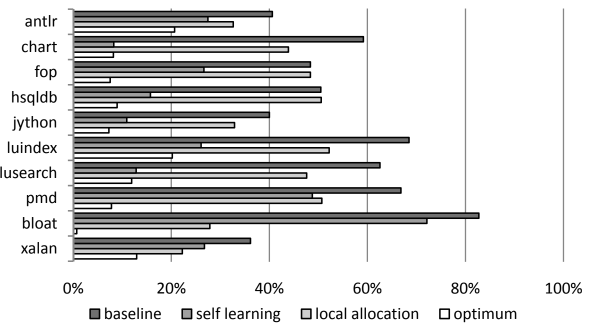

Figure 3: Remote access ratio comparison of four allocation strategies for all benchmarks assuming the ten hottest methods are executed by hardware accelerators.

4. EXPERIMENTAL RESULTS

For the evaluation of the techniques for data placement, we use the DaCapo benchmark suite [1], a famous suite for Java benchmarking that generates memory-intensive workloads. In an initial profiling run, we determine for each benchmark the ten hottest methods, i.e. those accounting for the largest execution time. For all our experiments, we assume that hardware-accelerated execution is used for each of these ten hottest methods.

4.1 Comparison of remote access ratios

Our first experiment is a global comparison of all four strategies for data placement over all benchmarks. We run the benchmark and instrument all memory accesses. Based on this measurement we then calculate the number of remote and local memory accesses during the complete run of the benchmark.

Figure 3 shows the remote access ratio for all four allocation strategies on the DaCapo benchmarks. Both the self-learning and the local allocation strategy reduce the remote access ratio significantly: the baseline approach suffers from 56% costly remote accesses on average, with local allocation this is reduced to 41%. Self-learning reduces the remote ac-

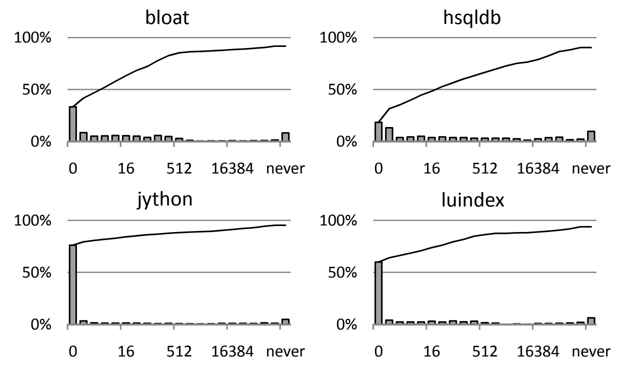

Figure 4: The self-learning strategy usually converges quickly.

cesses further to 28%. In the theoretical optimal case, when every object is allocated in the most suitable memory region, 11% of all memory accesses is a remote access.

The self-learning strategy can always reduce the communication cost compared to the baseline implementation, and, except for benchmarks `bloat` and `xalan`, performs better than local allocation. For `xalan` the difference between local allocation and the self-learning strategy is limited. In benchmark `bloat` lots of objects are allocated by the accelerated methods at the start of the program, before the self-learning algorithm can learn that these objects will be referenced more by the accelerator than the processor. This explains why the remote accesses ratio of self-learning for `bloat` is very close to the baseline approach.

4.2 Self-learning: how fast can it learn?

In a second experiment we measured the convergence of the self-learning allocation strategy. This strategy places objects close to where they were referenced the most in the past. After a sufficient length of time, this algorithm converges to the final (and optimal) allocation site.

For each creation site in each benchmark, we counted how many objects were created before the algorithm has converged. We clustered these creation sites according to this number of wrongly placed objects and weighted them by the total fraction of objects they represent. Figure 4 shows the resulting histograms and the cumulative curve for a representative selection of benchmarks. For example, in `jython` the optimal allocation site for 76% of the objects was main memory. Since the self-learning algorithm defaults here, these objects are allocated correctly from the start of the program. In Figure 4 this is shown as a bar of 76% at zero: for these creation sites zero objects are allocated incorrectly.

The behaviour of benchmark `hsqldb` is more complicated. Here, main memory is the optimal allocation site for only 18% of all objects. For other creation sites, the first object is placed in main memory but it is subsequently accessed more by the accelerator. Therefore the self-learning strategy allocates the second object in the accelerator's local memory. If the accelerator remains responsible for the majority of accesses to these objects, then the self-learning strategy will correctly place all subsequent objects in the accelerator's local memory —so in this case only the first object of each site was allocated incorrectly. For `hsqldb` these creation sites amount to 13% of all objects, as visible in the bar at one in Figure 4. At other creation sites, more than one object was allocated before accesses from one component clearly outnumber accesses from the other component. Finally, the last bar in the histogram shows that creation sites that never converge towards the optimal location amount to 10% of all objects in `hsqldb`.

In general we can conclude that the self-learning algorithm converges quickly. The fraction of objects allocated at creation sites for which the algorithm never converges, is never more than 10%.

5. CONCLUSIONS

Although application-specific hardware accelerators can significantly improve the performance of JVMs, communication cost often limits the speedup obtained in practice. In our hybrid architecture this cost is caused by the non-uniformity of access times to the distributed heap memory, formed by main memory and local memory of the accelerator. We propose several techniques that can find the optimal location for each Java object's data and thereby reduce the communication by up to 86% for the DaCapo benchmarks.

6. REFERENCES

[1] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of OOPSLA*, pages 169–190, Oct. 2006.

[2] A. Borg, R. Gao, and N. Audsley. A co-design strategy for embedded Java applications based on a hardware interface with invocation semantics. In *Proceedings of JTRES*, pages 58–67, 2006.

[3] E. A. Hakkennes and S. Vassiliadis. Multimedia execution hardware accelerator. *Journal of VLSI signal processing systems for signal image and video technology*, 28(3):221–234, 2001.

[4] R. Helaihel and K. Olukotun. Java as a specification language for hardware/software systems. In *Proceedings of ICCAD*, pages 690–697, 1997.

[5] E. Lattanzi et al. Improving Java performance using dynamic method migration on FPGAs. *International Journal of Embedded Systems*, 1(3):228–236, 2005.

[6] R. Lysecky, G. Stitt, and F. Vahid. WARP processors. *Transactions on Design Automation of Electronic Systems*, 11(3):659–681, July 2006.

[7] R. P. Maddimsetty et al. Accelerator design for protein sequence HMM search. In *Proceedings of ICS*, pages 288–296, 2006.

[8] E. M. Panainte, K. Bertels, and S. Vassiliadis. The MOLEN compiler for reconfigurable processors. *Trans. on Embedded Computing Sys.*, 6(1):6, 2007.

[9] C. Porthouse. Jazelle for execution environments. ARM Whitepaper, available online, May 2005.

[10] M. Schoeberl. A Java processor architecture for embedded real-time systems. *J. Syst. Archit.*, 54(1-2):265–286, 2008.

[11] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte. The MOLEN polymorphic processor. *IEEE Trans. on Computers*, 53(11):1363–1375, 2004.