Efficient Memory Management for Hardware Accelerated Java Virtual Machines

PETER BERTELS, WIM HEIRMAN, ERIK D'HOLLANDER,
and DIRK STROOBANDT
Ghent University

Application-specific hardware accelerators can significantly improve a system's performance. In a Java-based system, we then have to consider a hybrid architecture that consists of a Java Virtual Machine running on a general-purpose processor connected to the hardware accelerator. In such a hybrid architecture, data communication between the accelerator and the general-purpose processor can incur a significant cost, which may even annihilate the original performance improvement of adding the accelerator. A careful layout of the data in the memory structure is therefore of major importance to maintain the acceleration performance benefits.

This article addresses the reduction of the communication cost in a distributed shared memory consisting of the main memory of the processor and the accelerator's local memory, which are unified in the Java heap. Since memory access times are highly nonuniform, a suitable allocation of objects in either main memory or the accelerator's local memory can significantly reduce the communication cost. We propose several techniques for finding the optimal location for each Java object's data, either statically through profiling or dynamically at runtime. We show how we can reduce communication cost by up to 86% for the SPECjvm and DaCapo benchmarks. We also show that the best strategy is application dependent and also depends on the relative cost of remote versus local accesses. For a relative cost higher than 10, a self-learning dynamic approach often results in the best performance.

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

Additional Key Words and Phrases: Dynamic memory management, Java Virtual Machine, hardware acceleration

P. Bertels is supported by a Ph.D. grant from the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). This research is also related to the FlexWare project (IWT grant 060068) and the OptiMMA project (IWT grant 060831).
Authors' addresses: P. Bertels, W. Heirman, E. D'Hollander, and D. Stroobandt, ELIS Department, Ghent University, Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium; email: {peter.bertels, wim.heirman, erik.dhollander, dirk.stroobandt}@ugent.be.

DOI 10.1145/1562514.1562516 http://doi.acm.org/10.1145/1562514.1562516

ACM Reference Format:
Bertels, P., Heirman, W., D'Hollander, E., and Stroobandt, D. 2009. Efficient memory management for hardware accelerated Java virtual machines. ACM Trans. Des. Autom. Electron. Syst. 14, 4, Article 48 (August 2009), 18 pages.
DOI = 10.1145/1562514.1562516 http://doi.acm.org/10.1145/1562514.1562516

1. INTRODUCTION

Hardware accelerators or other application-specific coprocessors are used to improve the performance of computationally intensive programs. Significant speedups have been obtained for several application domains: multimedia [Vassiliadis et al. 2004; Eeckhaut et al. 2007; Lysecky et al. 2006], bio-informatics [Maddimsetty et al. 2006; Faes et al. 2006], and many other applications where small computational kernels with a sufficient amount of internal parallelism are used.

The first attempts for hardware/software codesign [Ernst et al. 1993; Gupta and De Micheli 1993] originate from software written in machine code languages (C and C++) or variants thereof. More recently, the same methodology has also been applied to Java programs [Helaihel and Olukotun 1997]. Two main directions can be identified in this domain: (i) acceleration of the Java Virtual Machine (JVM) itself and (ii) acceleration of specific methods of Java programs.

In this article, we concentrate on the latter approach because it uses the hardware only for those specific functions where a significant speedup can be obtained, as has been shown in several hardware implementations [Hakkennes and Vassiliadis 2001; Eeckhaut et al. 2007]. Less hardware-friendly methods are left in software. This approach leads to a hardware accelerated JVM which is described in Section 2 and contrasts to the first approach where often a significant amount of hardware resources need to be allocated for functionality such as scheduling or memory management . . .

A remaining advantage of accelerating the JVM as a whole is its complete transparency to the programmer. In the second approach, the programmer usually has to manage the communication and synchronization between the hardware accelerator and software methods. However, the JVM can be extended to manage this communication thereby again providing transparency. In this article, we use the adapted JVM of Faes et al. [2004] which can even, if the hardware accelerator is reconfigurable, move functionality dynamically from the general-purpose processor to the accelerator. Hardware execution is now an additional optimization step in the just-in-time compiler, where the hardware configuration can be loaded from a library [Borg et al. 2006] or could even be generated on-the-fly [Beck and Carro 2005; Lysecky et al. 2006].

For performance reasons both the main processor and the accelerator have their own local memory which are unified through Java's shared-memory model [Faes et al. 2007]. Our transparent hardware accelerated JVM manages memory accesses to all objects independent of their physical location. The location of the objects now has an important impact on the overall system performance. Since the accelerator is usually connected through a relatively slow

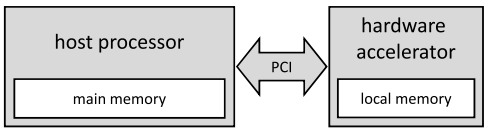

Fig. 1. Hybrid hardware platform consisting of a general-purpose host processor and an application-specific hardware accelerator.

communication medium, remote memory accesses are expensive and should thus be avoided as much as possible.

The placement of objects in the distributed Java heap is now an important task of the JVM. It 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 thus minimizing extraneous communication overhead. Solving this data placement problem is the main contribution of this article.

The object placement problem cannot be solved by a static analysis alone, as it can only estimate which data are private to a method conservatively. Moreover, 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 the general-purpose processor and the 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. Two strategies are distinguished: a profiling-based approach and a self-learning strategy. In the first approach all read and write memory accesses are counted during a distinct profiling phase. In subsequent executions, data will immediately be allocated at the optimized locations calculated based on the gathered statistics. The self-learning approach tries to estimate the usage patterns for each object based on measured patterns for previously allocated objects. These two techniques are compared to a baseline algorithm which does not take communication cost into account and a static technique for local memory allocation that 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% (49% on average) for the SPECjvm and DaCapo benchmarks (Section 4).

2. HARDWARE ACCELERATED JAVA VIRTUAL MACHINE

2.1 Host PC 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 (generally the more control-dominated parts).

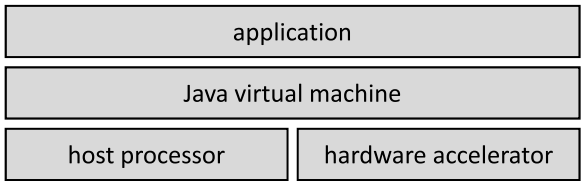

Fig. 2. The Java Virtual Machine hides the complexity of the underlying hybrid architecture from the application.

In our hybrid architecture, the processor and the accelerator both have their own local memory. However, the connection between the two components is realized by means of a relatively slow bus such as PCI, HyperTransport, . . . Therefore, the use of our approach is limited to algorithms with a high computation to communication ratio. For this class of applications a significant speedup is obtained by exploiting the massive parallelism available on FPGAs or ASICs [Panainte et al. 2007].

2.2 JVM as Hardware Abstraction Layer

We want to hide the complexity of managing the control flow and the communication between the accelerator and the host processor from the programmer. Also, if 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). In Faes et al. [2004], we have proposed a system where the JVM intercepts method calls for which a hardware equivalent is available and delegates 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 behavior of the bytecode in the corresponding Java method. The accelerator thus is stateless. At each invocation, both the parameters of the function and the corresponding state (the corresponding class for static methods, an object reference for virtual methods) need to be transferred to the hardware component. The JVM uses the same approach: Software objects encapsulate both state and function but at runtime state and function are separated. Static class information and Java bytecode are stored on a per-class basis and instance information (state) on a per-object basis. This equivalence between the hardware accelerator and Java methods is described in detail in Borg et al. [2006].

Method calls for which a hardware equivalent is available are intercepted by the JVM. The execution of the current thread is delegated to the accelerator unless the accelerator is not available (it may be in use by another

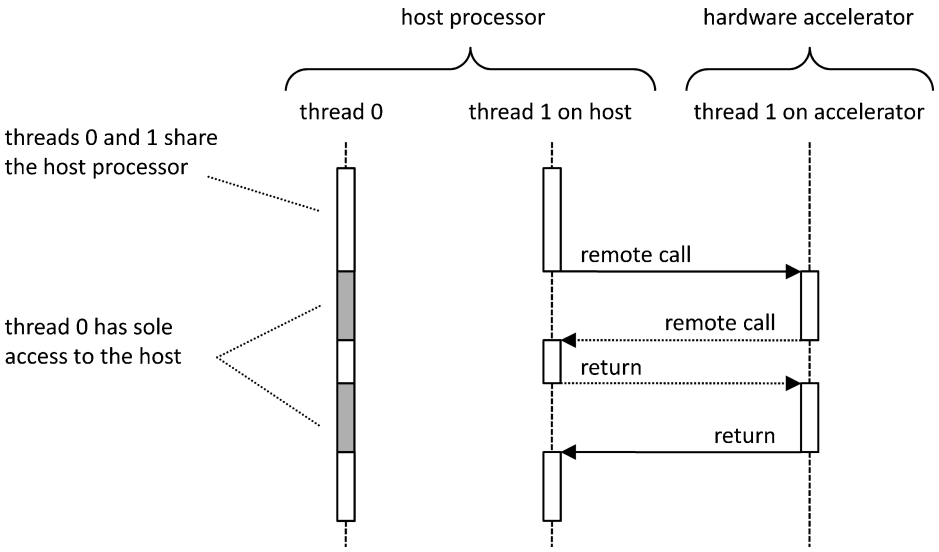

Fig. 3. Threads executed on the host processor can start the hardware accelerator which can perform a callback to the host when necessary, for example, for file input/output or for throwing exceptions.

thread), in which case the Java version of this method continues on the host processor.

The communication between the host processor and the hardware accelerator is based on remote calls. Figure 3 shows two of these calls. The first remote call, represented by a solid line, is initiated by the thread executed on the host and starts the accelerator method. The parameters of the method are passed by reference. The host processor suspends execution of the current thread while the hardware accelerator is busy. Meanwhile, the host can continue executing other threads. When functionality called by this accelerated method has no hardware equivalent or is simply impossible to implement in hardware, the accelerator can rely on the host processor to execute this specific functionality via a callback mechanism. This is typically used for file input/output or for throwing exceptions. Such a callback is depicted in Figure 3 as a dotted line.

2.3 Shared-Memory Model

Our hybrid architecture uses a shared-memory model, which 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, including those held by the accelerator [Faes et al. 2005]. Whether new objects are placed in main memory or in the accelerator's local memory should depend on the access patterns. 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.

We assume that, if the processor caches its accesses to main memory, coherence is 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, optimizing the object placement will reduce the communication overhead and may reduce cache requirements for remote addresses.

In this article, we focus on the (optimal) allocation of objects to main memory or the accelerator's memory. We do not consider the possibility of moving objects from one memory to the other as this introduces an additional overhead. Additionally most objects do not live long enough to warrant relocation.

3. STRATEGIES FOR DATA PLACEMENT

As a general rule, objects should be placed close to the component (host processor or hardware accelerator) that references them most. This way, a large fraction of memory accesses will be local, minimizing 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.

In most cases, one creation site allocates objects of the same type and with very similar usage patterns. Some specific software patterns break this general rule. For example, in class factories a single creation site creates objects of different types which are used in very different ways and in 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.

In this article, we 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, whether the allocation site is adaptive or fixed, and whether it is based on runtime or profile information.

3.1 Baseline Algorithm

A first approach is our baseline algorithm in which all objects are allocated in main memory as most methods are executed on the processor. This algorithm takes no account of the hybrid nature of our architecture. 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.

3.2 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 allocation 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 other strategies with.

This strategy needs a separate profiling phase for gathering the global data usage patterns per creation site [Bertels et al. 2008]. We measured the profile for a specific run of the application. Alternatively, usage patterns can be accumulated during several profiling runs. The optimal placement strategy incurs no runtime overhead since the allocation decision for each creation site is fixed.

3.3 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 and therefore implementation of local allocation is straightforward and incurs no runtime overhead. Moreover, no separate profiling phase is needed.

3.4 Self-Learning Allocation

In self-learning allocation, 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

Table I. Basic Properties of Data Placement Strategies

Strategy	Decision	Usage Patterns	Adaptive	Runtime Overhead
baseline	compile time	no	no	no
optimal allocation	compile time	yes	no	no
local allocation	runtime	no	yes	no
self-learning	runtime	yes	yes	yes

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 accesses these objects the most up to now. New objects created at this creation site will be allocated in the memory corresponding 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 (Section 3.2). The allocation decision for new objects in the self-learning algorithm will therefore converge towards the decision as in the optimal placement. This convergence usually happens very quickly, as shown in Section 4, and therefore this strategy still results in a strongly reduced remote access ratio.

4. EXPERIMENTAL RESULTS

For the evaluation of the techniques for data placement, we use the SPECjvm [Standard Performance Evaluation Corporation 2008, 1998] and DaCapo benchmark suites [Blackburn et al. 2006]. We excluded the duplicates in the SPECjvm98 and SPECjvm2008 suites, retaining the most recent version. Benchmarks `sunflow` and `crypto.signverify` were also excluded because they create less than 100 objects.

In an initial profiling run, we determine for each benchmark the ten hottest methods, namely those accounting for the largest execution time. For all our experiments, we assume that a hardware equivalent for each of these ten hottest methods is available. The simulated hybrid architecture will dynamically decide whether to execute each method on the processor or on the accelerator. For ten hardware-accelerated methods this leads to 2^{10} possibilities for partitioning the functionality. In our experiments, we used only 10 of these 1024 possible partitionings, and each of these delegates n methods to the hardware accelerator, where n ranges from 1 to 10. For each of these 10 partitionings, we evaluated each allocation strategy.

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. In this experiment, all 10 hottest methods are assumed to be running on the hardware accelerator.

This means that in the baseline approach, where all objects are allocated in main memory, all memory accesses from hardware accelerated methods are

remote accesses. Once some objects are allocated in hardware memory, accesses to these objects from the accelerator will no longer be remote while in contrast accesses by the processor are now remote. For the optimal allocation, it is evident that the total number of remote accesses is always smaller than in the baseline approach. For the other strategies, this cannot be guaranteed although in most cases a significant reduction is obtained.

Figure 4 shows the remote access ratio for all four allocation strategies on the SPECjvm and DaCapo benchmarks. In this experiment we placed all ten hottest methods in hardware. Four classes of applications can be distinguished.

For applications in class I, II, and III, we can see the expected results: Optimal data placement performs best, baseline performs worst. Self-learning and local allocation are between the two extremes. Self-learning performs better than local allocation for applications in class II, local allocation is better than self-learning in class III. In class I, both have similar behavior.

The outlier, `209db`, is in class IV: In this application lots of objects are created by the accelerators, while they are heavily used by the main processor. Specifically, the method `Database.set_index()` creates arrays (of type `Entry[]`) for indexing the database. These objects are heavily used by methods executed on the host processor, but they are never used by the hardware accelerator. Therefore local allocation performs very badly. The baseline approach is very efficient by allocating all objects, including these `Entry[]` objects, in main memory. The self-learning and optimal strategies perform even slightly better because these approaches benefit from a few other objects which are used only internally in a hardware accelerated method, for example, the `StringBuffer` in `ValidityCheckOutputStream.strip(int, InputStream)`.

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 where they were referenced most in the past. After a sufficient length of time, this algorithm converges to the final allocation site.

For each creation site in each benchmark, we count how many objects were created before the algorithm has converged. This number defines the *pace of convergence* of each creation site. For a representative selection of benchmarks this convergence is shown in Figure 5. On the horizontal axis, we have clustered the creation sites according to their pace of convergence. The bars on the vertical axis show how many objects were created by each cluster of creation sites, as a relative fraction of all objects in the application. This indicates the importance of each cluster. Figure 5 also shows a cumulative curve for the relative fraction of objects.

For example, in benchmark `209db` the optimal allocation site for almost all objects was main memory, including for those objects created by the accelerator. Since the self-learning algorithm defaults to main memory, all objects are allocated correctly from the start of the program. In Figure 5 this is shown as a bar of 99.96% at zero: For almost all creation sites zero objects are allocated incorrectly. For this reason `209db` was also an outlier in Figure 4.

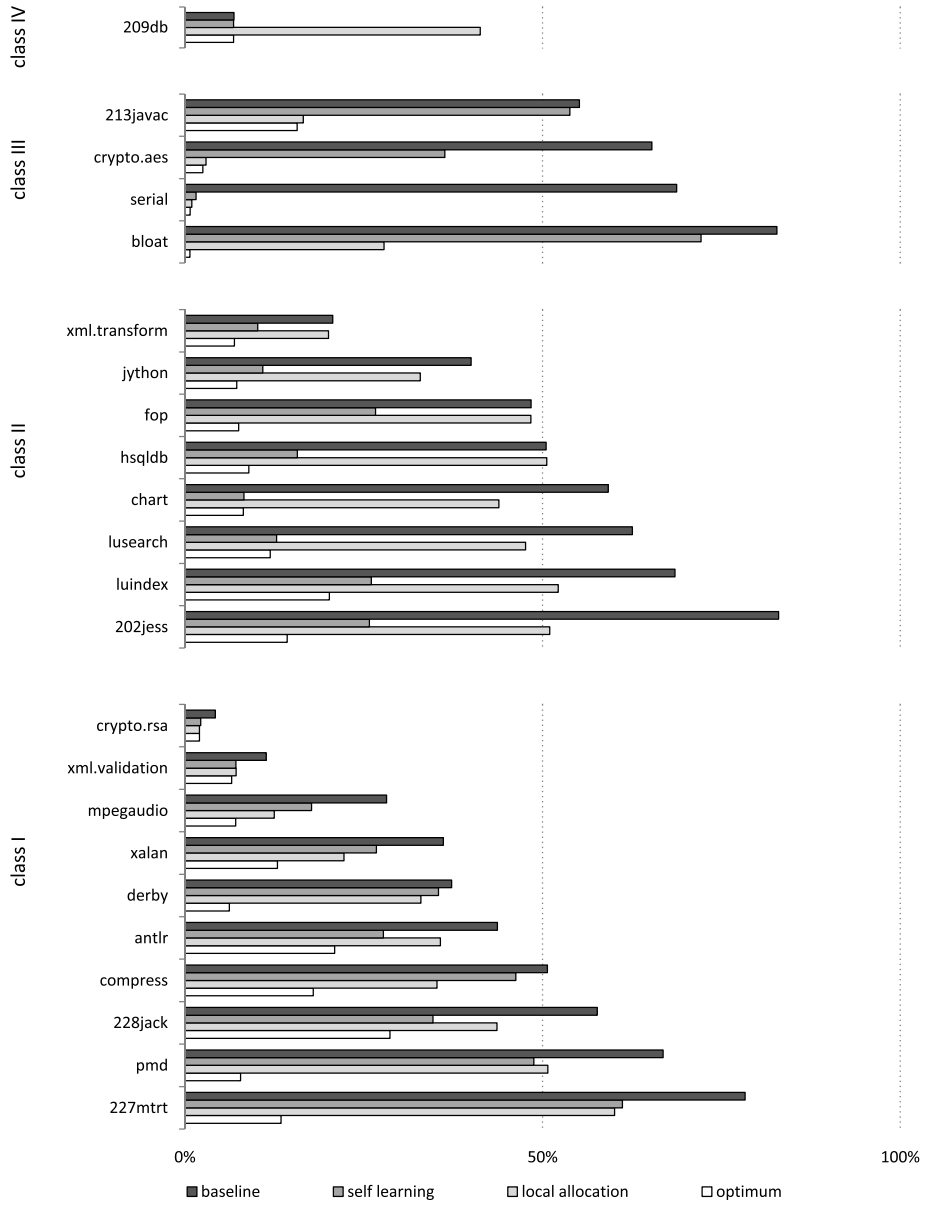

Fig. 4. Remote access ratio comparison of four allocation strategies for all benchmarks assuming the ten hottest methods are executed by hardware accelerators.

The behavior of benchmark `227mtrt` is more complicated. For 10% of all objects, the optimal allocation site is main memory, just as for benchmark `209db` this results in an 10% bar at zero in the histogram. For some creation sites, the first object is placed in main memory but is subsequently accessed more by the accelerator. Therefore the self-learning strategy allocates the second object in

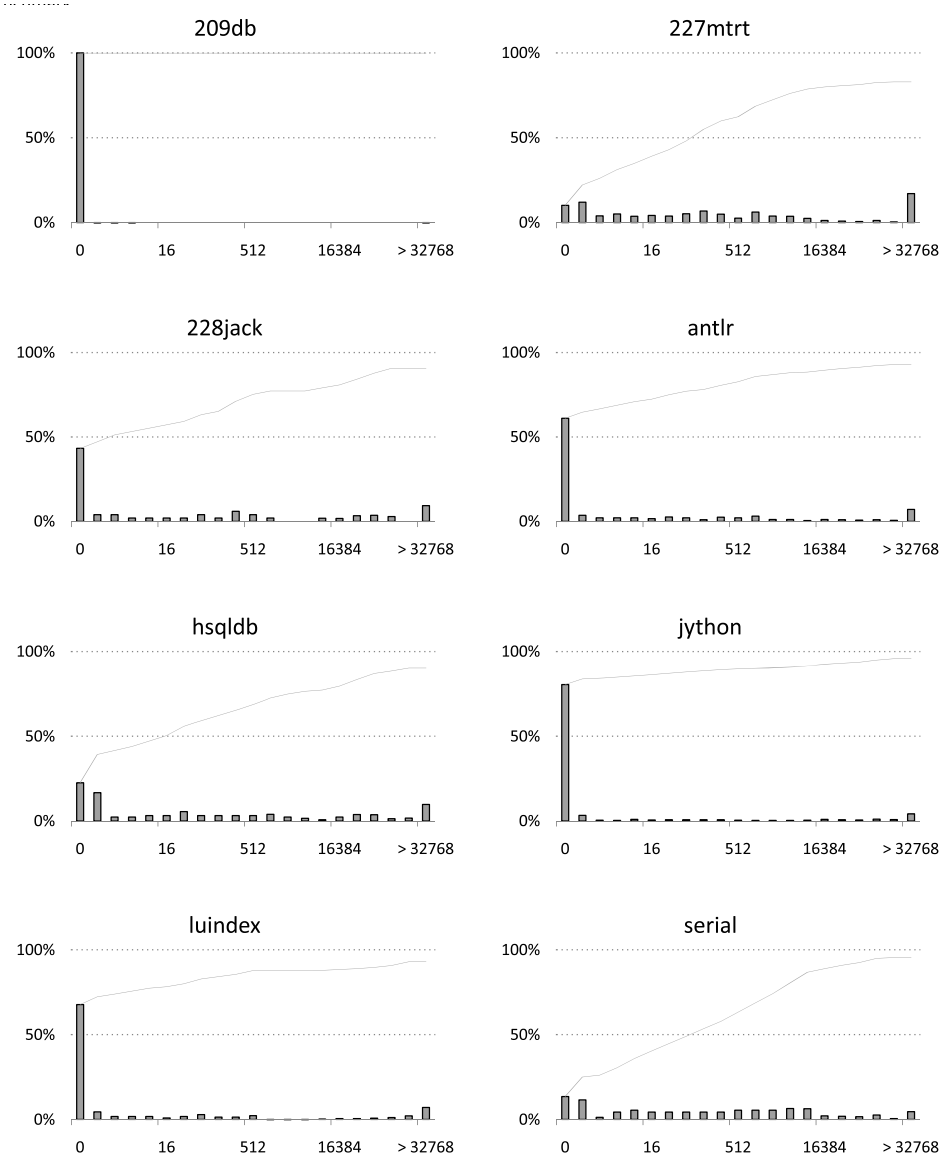

Fig. 5. On the horizontal axis, we have clustered the creation sites according to their *pace of convergence*, defined as the number of objects created at this creation site before convergence. The bars on the vertical axis show how many objects were created by each cluster of creation sites, as a relative fraction of all objects in the application, indicating the importance of each cluster. Each graph also contains a cumulative curve for the relative fraction of objects. For most benchmarks, self-learning learns very fast for the most important creation sites.

the accelerator's local memory. If the accelerator stays responsible for the major part of all accesses to these objects, then the self-learning strategy will place all subsequent objects in the accelerator's local memory, which is indeed the optimal allocation site for objects from these creation sites. So only the first object of each site was allocated incorrectly. For benchmark `227mtrt` these creation sites amount to 12% of all objects as visible in the bar at one in Figure 5. For 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 (or extremely slowly) converge towards the optimal location amount to 17% of all objects in `227mtrt`.

In general, we can conclude from Figure 5 that the self-learning algorithm converges rather quickly. The fraction of objects allocated at creation sites which never converge is never more than 20% (this maximum is reached for benchmark `compress`).

4.3 Self-Learning in Practice

The previous sections have shown that self-learning performs quite well for most benchmarks. Through its dynamic and adaptive nature, this algorithm can also be used in situations where the memory behavior of the application is not known at compile-time. This is the case for applications which have a data-dependent behavior or in applications where the decision whether to execute functionality on the general-purpose processor or on a specific hardware accelerator is taken at runtime. This last option includes the case where hardware-accelerators are constructed on-the-fly as an extension of the traditional just-in-time compilation process.

The only drawback of the approach is the fact that this technique is expensive. All memory accesses have to be profiled and for each access, the counters for the relevant objects need to be updated. This increases the execution time by a factor of ten or more. Clearly this is unacceptable when compared to a possible performance improvement that is for most benchmarks more modest.

The cost of memory access profiling can be greatly reduced by using a sampled approach. In this case, only a certain fraction of memory accesses is instrumented.

Since the patterns with the largest influence on performance are those with the largest volume of communication, and as those patterns evolve rather slowly, taking a small sample of these memory accesses should not significantly change the behavior of the self-learning allocation strategy for most of the benchmarks. This is evident from Figure 6. This figure shows the ratio of remote memory accesses for a representative selection of benchmarks using the self-learning allocator, comparing the fully instrumented version with five levels of sampling ranging from 1/10 to 1/100,000. Each bar represents the average amount of remote accesses over 100 uniformly random samplings, where error bars mark the maximum and minimum values.

For `209db`, both the baseline and self-learning approaches perform close to the optimum. Here, sampling barely influences this result. Benchmark `serial`

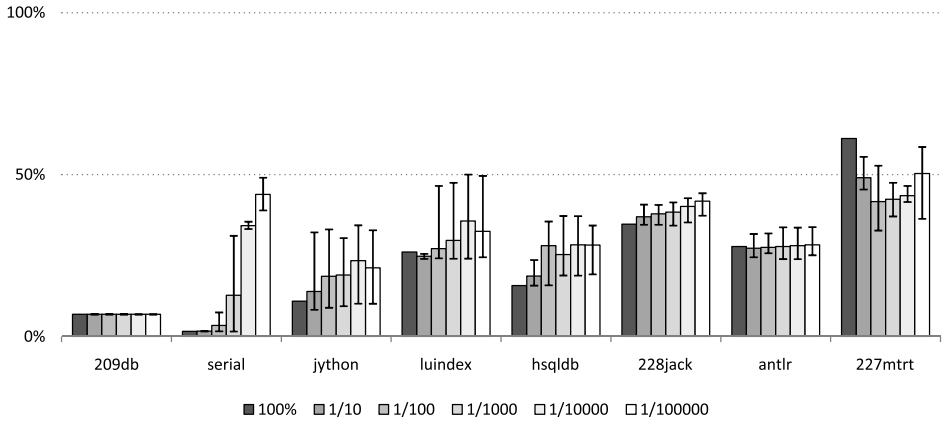

Fig. 6. Ratio of remote memory accesses for a representative selection of benchmarks using the self-learning strategy, comparing the fully instrumented version with five levels of sampling ranging from 1/10 to 1/100,000. Sampling increases the remote access ratio but the final performance is still tolerable.

belongs to class III where local allocation outperforms the self-learning approach. With a sampling ratio of up to 1/100 the self-learning strategy remains competitive, smaller sampling ratios quickly decrease performance.

For benchmarks in class II (`jython`, `luindex` and `hsqldb`) the self-learning strategy resulted in the lowest remote access ratio, close to the optimum. Unfortunately, the profiling overhead precluded actual performance improvement in a realistic setting. However, a sampling ratio of up to 1/100,000 keeps the self-learning allocator's performance intact, but greatly reduces the profiling overhead.

The final set of benchmarks (`228jack`, `antlr` and `227mtrt`), belonging to class I, show a similar result. Up to a sampling ratio of 1/100,000 the change in performance is minimal. Benchmark `227mtrt` is a strange case in that performance actually increases for sampling ratios up to 1/10,000. The self-learning strategy is eluded when all objects allocated at a specific allocation site have different usage patterns: Some of these objects are used more by the accelerator, others are used more by the host processor, but their usage patterns are clustered at the same allocation site. Therefore differences between these objects cannot adequately be taken into account. As described in Section 3, this behavior sometimes results from specific coding styles, such as a class factory. However, in case of `227mtrt`, the problem is not caused by a class factory. Benchmark `227mtrt` is a raytracer. At several allocation sites, this application allocates `Point` objects which are used by methods on both sides of the hardware/software boundary. Which method uses which specific `Point` obviously depends more on the image that is raytraced (data dependency) than it depends on the allocation site.

Overall we can conclude that a sampling ratio of 1/1,000 or even 1/10,000 does not significantly increase the remote access ratio. However, the profiling overhead, which may amount to about 100 times the original execution time, is now also reduced by a factor of 10,000, to just 1% of the execution time.

This enables a practical implementation of the self-learning strategy, where the performance gained by reducing the number of expensive remote accesses greatly outweighs the remaining profiling overhead.

4.4 Impact on the Execution Time

From the previous sections it is clear that our object placement strategies drastically reduce the number of remote accesses. We will now show that this also leads to a significant reduction in execution time of the benchmarks.

Because the impact on the execution time depends on the relative cost of remote memory accesses over local memory accesses, we plotted the execution time as a function of this relative cost. For one benchmark of each class in Figure 4 we compare the execution time of the baseline approach, the local allocation strategy, and two sampled implementations of the self-learning strategy in Figure 7.

For `serial`, a benchmark of class III with 69% of nonlocal accesses in the baseline approach, the solid line in the graphs shows that the execution time increases with a factor of 2.85 when remote memory accesses are 10 times as costly as local accesses. In an architecture where the relative cost of remote accesses is 100, the execution time even increases by a factor of 21, while with local allocation (dotted line), the best strategy for `serial`, there is only an increase of 18%. The self-learning strategy (dashed lines) leads to a similar reduction of the remote accesses as local allocation but it incurs an overhead for runtime instrumentation of the program. In Figure 7, we assumed an overhead of a factor of ten when all memory accesses are instrumented or a factor of two when 1 out of 10 accesses is measured. The form of the curve of the self-learning strategy with sampling ratio 1/10 is similar to that of local allocation, because of the similar reduction of the number of remote accesses, but the instrumentation overhead still makes the program two times slower. The instrumentation overhead becomes negligible for 1/10,000 sampling, but then the resulting reduction in remote accesses is worse, as explained in Section 4.3.

As in the previous experiments, benchmark `209db` is again a special case: The baseline approach performs best with an increase in execution time of only a factor of three for the most extreme scenario where remote accesses cost 100 times more than local accesses. The local allocation leads to an increase of a factor of 13 in this scenario. Self-learning reduces the fraction of remote accesses to the same level as the baseline approach. Therefore the curve for the 1/10,000 sampled implementation of self-learning coincides with the baseline approach. For a sampling ratio of 1/10 the instrumentation overhead can never be recovered.

For benchmarks in class II, our self-learning strategy performs far better than local allocation. `luindex` is an example of this class. Self-learning performs best in reducing the fraction of remote memory accesses. With a sampling ratio of 1/10,000 this results in a speedup from a relative cost of remote accesses of 2 onwards. Instrumenting more memory accesses (sampling ratio of 1/10) performs even better when the relative cost of remote accesses exceeds a factor of 50.

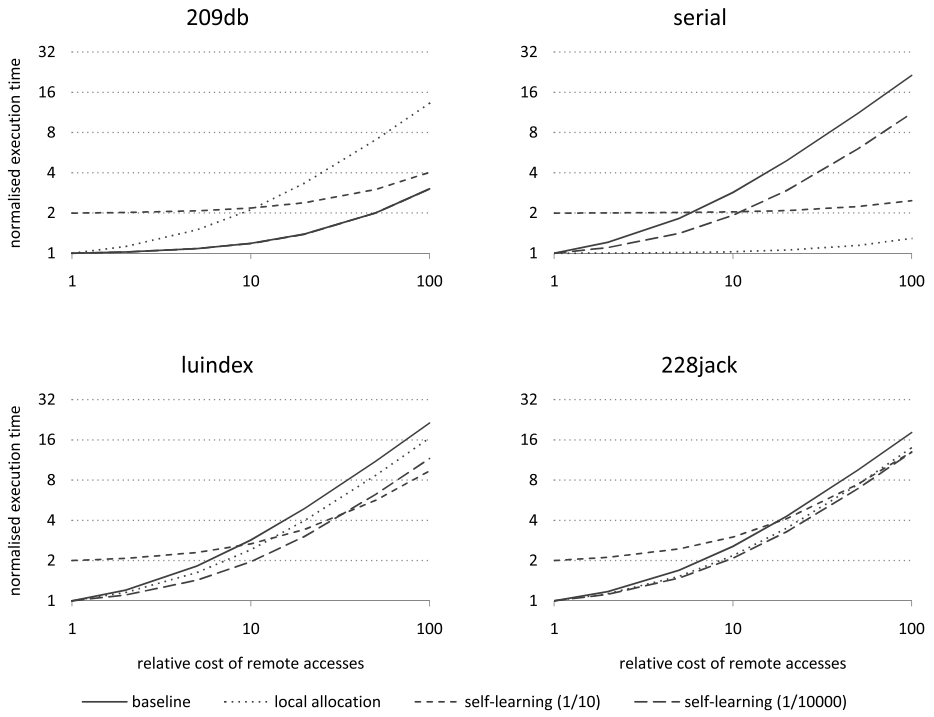

Fig. 7. Normalized execution time as a function of the relative cost of remote memory accesses over local memory accesses.

`228jack` is an example of class I where both self-learning and local allocation lead to a similar reduction of remote memory accesses. In the baseline approach the execution time increases with a factor of 18 for the most extreme scenario, while the other approaches limit this increase to a factor of 13. Allocating the objects in the appropriate memory thus leads to a performance increase of about 27%.

5. RELATED WORK

Several approaches have been proposed to extend a general-purpose processor with application-specific hardware accelerators. Although the exploitation of parallelism on the hardware accelerator is similar in those approaches, the way the accelerator communicates to the host processor is quite different. In some of these hybrid computing systems, the MOLEN processor [Vassiliadis et al. 2004] and the WARP processor [Lysecky et al. 2006], the host processor and the accelerator are directly connected to a shared memory. This contrasts to our approach and the approach described in Lattanzi et al. [2005] where the shared memory is distributed and the cost of these accesses is highly nonuniform. In such an architecture with nonuniform distributed memory, our strategies for dynamic memory allocation are indispensable.

Lattanzi divides the memory in three parts: a dual-port memory accessible by both components and two single-port memories which take the form of local

memory for one of these components. Java objects that are used by both components need to be allocated in the dual-port memory. In a dynamic environment where functionality can float from the general-purpose processor to the accelerator, this leads to a situation where all objects used by methods which are candidates for hardware acceleration need to be allocated in the dual-port memory, even when these candidate methods are currently executed on the processor. In the limit this requires the complete memory to be dual-port, annihilating the benefits of using local memory.

The components in our hybrid architecture have access to both their own local memory and, remotely, the local memory of the other component. We have shown that local allocation and the self-learning strategy can significantly reduce the number of remote accesses and, consequently, the involved communication cost, also eliminating the dual-port memory needed in Lattanzi's approach.

The strategies proposed in this article significantly reduce the communication cost in our hybrid architecture. These techniques prove their value in the static case when partitioning of functionality between the processor and the accelerator is fixed. They become indispensable when we move further to a more dynamic hybrid architecture with a reconfigurable accelerator where the JVM generates hardware on-the-fly. Runtime hardware generation for random functions is not possible yet, but several approaches have proven successful for fine-grain functionality. A practical implementation of this can be found in the WARP processor [Lysecky et al. 2006], a new processing architecture that consists of a microprocessor and an FPGA. The execution of a software binary on the microprocessor is profiled in order to detect hot code fragments. For these fine-grain, functional blocks, a hardware accelerator is generated on the FPGA. A similar methodology has been applied for Java programs [Beck and Carro 2005].

These approaches are too fine-grain to be directly incorporated in our hybrid architecture, but the results thus far indicate that dynamic memory allocation strategies for such an architecture will even gain importance in the future.

6. CONCLUSIONS

Although application-specific hardware accelerators can significantly improve the performance of Java Virtual Machines, communication cost often limits the speedup obtained in practice. In our hybrid architecture this cost is caused by the nonuniformity 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 SPECjvm and DaCapo benchmarks. This leads to a reduction in execution time of a factor of 1.64 on an architecture where remote memory accesses are 20 times slower than local accesses. In a first class of benchmarks, local allocation performs best and is moreover very efficient to implement, involving no runtime overhead. For a second class of benchmarks, our self-learning algorithm can further reduce communication by dynamically choosing allocation sites based on runtime instrumentation of memory accesses. The overhead of

this instrumentation can be kept acceptable through the use of sampling: A sampling ratio of up to 1/10,000 can be applied while still significantly reducing the communication cost. The performance gained by reducing the number of remote accesses greatly outweighs the remaining profiling overhead.

ACKNOWLEDGMENTS

The authors would like to thank K. Bruneel and H. Devos and the anonymous reviewers for their valuable suggestions.

REFERENCES

BECK, A. C. S. AND CARRO, L. 2005. Dynamic reconfiguration with binary translation: Breaking the ILP barrier with software compatibility. In *Proceedings of the 42nd Annual Design Automation Conference (DAC)*. ACM, New York, 732–737.

BERTELS, P., HEIRMAN, W., AND STROOBANDT, D. 2008. Efficient measurement of data flow enabling communication-aware parallelisation. In *Proceedings of the International Forum on Next-Generation Multicore/Manycore Technologies (IFMT)*. ACM, New York, 1–7.

BLACKBURN, S. M., GARNER, R., HOFFMAN, C., KHAN, A. M., MCKINLEY, K. S., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., HIRZEL, M., HOSKING, A., JUMP, M., LEE, H., MOSS, J. E. B., PHANSALKAR, A., STEFANOVIĆ, D., VANDRUNEN, T., VON DINCKLAGE, D., AND WIEDERMANN, B. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'06)*. ACM Press, New York, 169–190.

BORG, A., GAO, R., AND AUDSLEY, N. 2006. A codesign strategy for embedded Java applications based on a hardware interface with invocation semantics. In *Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*. ACM, New York, 58–67.

EECKHAUT, H., DEVOS, H., LAMBERT, P., DE SCHRIJVER, D., VAN LANCKER, W., NOLLET, V., AVASARE, P., CLERCKX, T., VERDICCHIO, F., CHRISTIAENS, M., SCHELKENS, P., VAN DE WALLE, R., AND STROOBANDT, D. 2007. Scalable, wavelet-based video: From server to hardware-accelerated client. *IEEE Trans. Multimedia 9*, 7, 1508–1519.

ERNST, R., HENKEL, J., AND BENNER, T. 1993. Hardware-software cosynthesis for micro-controllers. *IEEE Des. Test Comput. 10*, 4, 64–75.

FAES, P., CHRISTIAENS, M., BUYTAERT, D., AND STROOBANDT, D. 2005. FPGA-aware garbage collection in Java. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 675–680.

FAES, P., CHRISTIAENS, M., AND STROOBANDT, D. 2004. Transparent communication between Java and reconfigurable hardware. In *Proceedings of the 16th IASTED International Conference Parallel and Distributed Computing and Systems*, T. Gonzalez, Ed. ACTA Press, Cambridge, MA, 380–385.

FAES, P., CHRISTIAENS, M., AND STROOBANDT, D. 2007. Mobility of data in distributed hybrid computing systems. In *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 386.

FAES, P., MINNAERT, B., CHRISTIAENS, M., BONNET, E., SAEYS, Y., STROOBANDT, D., AND VAN DE PEER, Y. 2006. Scalable hardware accelerator for comparing DNA and protein sequences. In *Proceedings of the 1st International Conference on Scalable Information Systems (InfoScale'06)*. ACM, 33.

GUPTA, R. K. AND DE MICHELI, G. 1993. Hardware-software cosynthesis for digital systems. *IEEE Des. Test Comput. 10*, 3, 29–41.

HAKKENNES, E. A. AND VASSILIADIS, S. 2001. Multimedia execution hardware accelerator. *J. VLSI Signal Process. Syst. Signal Image Video Technol. 28*, 3, 221–234.

HELAIHEL, R. AND OLUKOTUN, K. 1997. Java as a specification language for hardware/software systems. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. IEEE Computer Society, 690–697.

LATTANZI, E., GAYASEN, A., KANDEMIR, M., VIJAYKRISHNAN, N., BENINI, L., AND BOGLIOLO, A. 2005. Improving Java performance using dynamic method migration on FPGAs. *Int. J. Embed. Syst. 1*, 3, 228–236.

LYSECKY, R., STITT, G., AND VAHID, F. 2006. WARP processors. *Trans. Des. Autom. Electron. Syst. 11*, 3, 659–681.

MADDIMSETTY, R. P., BUHLER, J., CHAMBERLAIN, R. D., FRANKLIN, M. A., AND HARRIS, B. 2006. Accelerator design for protein sequence HMM search. In *Proceedings of the 20th Annual International Conference on Super-Computing (ICS '06)*. ACM, New York, 288–296.

PANAINTE, E. M., BERTELS, K., AND VASSILIADIS, S. 2007. The MOLEN compiler for reconfigurable processors. *Trans. Embed. Comput. Syst. 6*, 1, 6.

STANDARD PERFORMANCE EVALUATION CORPORATION. 1998. Java Virtual Machine benchmarks (SPECjvm1998).

STANDARD PERFORMANCE EVALUATION CORPORATION. 2008. Java Virtual Machine benchmarks (SPECjvm2008).

VASSILIADIS, S., WONG, S., GAYDADJIEV, G., BERTELS, K., KUZMANOV, G., AND PANAINTE, E. M. 2004. The MOLEN polymorphic processor. *IEEE Trans. Comput. 53*, 11, 1363–1375.

Received November 2008; revised March 2009; accepted April 2009