View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Ghent University Academic Bibliography

How Java Increases Flexibility & Run-Time Efficiency of MPSoC Systems

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

***Abstract*—Embedded software has to meet multiple requirements: power efficiency, soft real-time performance, ... but also flexibility and adaptability to ever changing functional requirements. Heterogeneous MPSoC systems offer an interesting platform to meet these requirements. We propose to run a Java Virtual Machine (JVM) on the embedded, general-purpose processor to add the necessary flexibility and adaptability to the system. The techniques proposed in this paper allow the JVM to dynamically partition the application and map the partitions at run-time to the suitable system components and also to reconfigure the system when needed. In doing this, the JVM continuously monitors the communication cost, often a limiting factor in MPSoC design. In previous work [3], [8] we have shown that the JVM can dynamically switch between executing threads on the general-purpose processor or on specific hardware blocks and that it can reduce the communication overhead in the system by up to 86%. In this paper we extend our approach to a reconfigurable platform in which the JVM also decides on the scheduling and mapping of functionality.**

***Index Terms*—MPSoC, run-time techniques, Java, partitioning, mapping, scheduling**

I. Introduction

Embedded software has to meet multiple and often stringent requirements: power efficiency, soft real-time performance, etc. On the other hand it needs to be flexible and adaptive to ever changing functional requirements in a world with very short time-to-market.

Because these stringent requirements cannot be met with general-purpose processors only, heterogeneous MPSoC platforms are the name of the game in modern embedded systems. To address the need for flexibility, the trend at the moment is away from very specific and dedicated platforms towards more generic solutions for a broader class of applications, e.g. in the wireless or in the multimedia domain.

We propose to run a Java Virtual Machine (JVM) on the embedded, general-purpose processor which is the main component of most of these platforms. This JVM acts as an abstraction layer for the underlying hardware. The techniques proposed in this paper allow the JVM to dynamically partition the application and map the partitions at run-time to the suitable system components. Both internal communication overhead and performance requirements (deadlines), are taken into account.

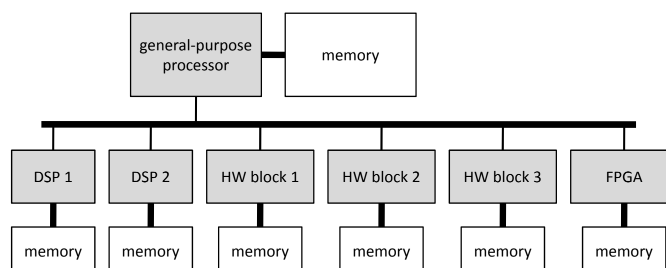

Fig. 1. The MPSoC platform considered in this paper, consists of a general-purpose host processor and several coprocessors, ranging from general Digital Signal Processors to application-specific hardware accelerators and reconfigurable logic.

Previously we have shown [3] how the JVM can dynamically minimise communication overhead by optimising memory allocation based on profiling information. In this paper, these techniques are extended for run-time partitioning and mapping of the Java application to a distributed platform.

We build on the JVM developed in [8] and extend it to make it fully aware of the underlying platform. We show how the JVM can autonomously reach close to optimal performance for a given application. Our approach enables efficient use of resources without need for specific compilation for a given platform. This way optimal flexibility is guaranteed at each functional update of the software.

This paper is organised as follows: Section II gives an overview of the envisioned MPSoC platform and how the JVM fits in this platform. In Section III we present profiling-based communication-aware mapping and scheduling techniques that will solve the communication bottleneck in the system. Finally, Section V concludes this paper.

II. Virtualised Approach to MPSoCs

A. MPSoC with central general-purpose processor

In this work, we use the classical concept of coprocessing. The MPSoC platform consists of a general-purpose processor and one or several application-specific hardware accelerators or DSP coprocessors or FPGA with reconfigurable hardware accelerators (Figure 1). In this paper we will use the term accelerator for all these coprocessing blocks. The accelerators execute a small but computationally intensive part of the

Java application, while the general-purpose processor executes the remainder of the application, that is in general the more control-dominated parts).

On this MPSoC platform, the main processor and these accelerators all have their own local memory which is connected to them with a high-bandwidth communication channel. The interconnect network which connects all these accelerators with the main processor is significantly slower and forms the main bottleneck that we will solve with the techniques presented in Section III.

Although our solution drastically reduces the communication overhead, our approach is still limited to applications with specific computational kernels with a high computation to communication ratio. Several authors have reported significant speedups for such computational kernels with specific hardware accelerators on FPGAs or ASICs [13].

B. JVM as hardware abstraction layer

We want to hide the complexity of managing the control flow and the communication between all these accelerators and the main processor from the programmer. Moreover the JVM can dynamically move functionality from the main processor to one of the reconfigurable FPGAs (Section III). This is all possible when we consider the JVM to be an abstraction for the underlying hardware. Faes has proposed a system where the JVM intercepts method calls for which a hardware equivalent is available and delegates execution to the appropriate accelerator [10]. This adapted JVM also enables the accelerators to access objects on the Java heap memory which is distributed between all physical memory blocks in the MPSoC.

In this concept, all accelerators form an integral part of the JVM while being invisible to the Java application itself. Therefore, we need to properly define an equivalence between the functionality of each component and software concepts in the Java language. In our approach, all accelerators encapsulate the functional behaviour of the bytecode in the corresponding Java method. This equivalence between the hardware accelerator and Java methods is described in detail in [5], [8].

The JVM intercepts all method calls for Java methods which are acceleratable. The execution of the current thread is delegated to the accelerator unless the accelerator is not available —it may be in use by another thread— in which case the Java version of this method continues execution on the main processor.

The communication between the general-purpose processor (GP) and one of the accelerators is based on remote calls as shown in Figure 2. The first remote call, represented by a solid line, is initiated by the thread executed on the GP and starts the hardware accelerator (HW). The GP suspends execution of the current thread while the HW is busy. Meanwhile, the GP can continue executing other threads. When functionality called by the accelerator 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 2 as a dotted line.

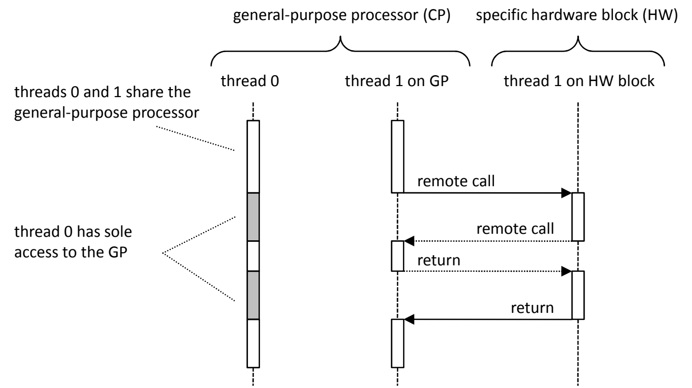

Fig. 2. Threads executed on the general-purpose processor (GP) can start one of the accelerators which can perform a callback to the GP when necessary, e.g. for file input/output or for throwing exceptions.

C. Shared-memory model

This MPSoC platform uses a shared-memory model, which allows the main processor and all accelerators to access all objects. The Java heap is distributed between the main memory of the GP and each accelerator's local memory. The garbage collector is extended to account for objects and references in all memories, including those held by accelerators [9]. Whether new objects are placed in main memory or in one of memories local to an accelerator, should depend on the access patterns. This is exactly the focus of our algorithm for communication-aware data placement which is described in [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.

Besides the close to optimal allocation of objects to main memory or to one the local memories, which we solved in [3], this paper focuses on the relocation of functionality between the general-purpose processor and one of the accelerators (Section III).

III. Communication-aware Mapping and Scheduling

A. Just-in-Time Compilation

The general idea of our communication-aware mapping and scheduling is an extension of the traditional Just-in-Time (JIT) compilation in the JVM. JIT compilation was first used in a very efficiënt implementation of Smalltalk [7], but is now commonly adopted by most virtual machines, e.g. the Sun's Java Hotspot compiler and several others [1] even in the embedded world [15], [12].

With conventional software JIT, the JVM starts interpreting the Java bytecodes and while interpreting, the JVM profiles the occurrences of all basic blocks in the code. Basic blocks which are frequently executed will then be intercepted an compiled to

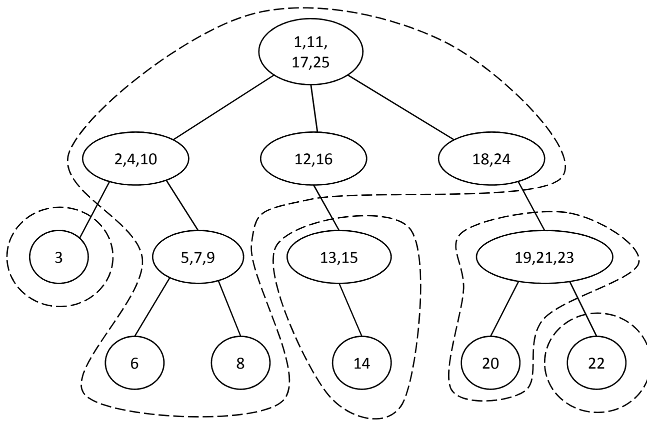

Fig. 3. The callgraph of a sample application with dotted lines indicating where this graph is cut to form separate partitions which can be mapped to accelerators of the MPSoC platform.

machine code. In most implementations this is a staged process with several steps of compilation, ranging from a very simple and basic compilation step which is very fast but results in relatively slow code, to an aggressive compilation step which takes more time to compile but returns excellent and very efficient machine code.

In this paper we want to extend this idea with an extra compilation stage which involves hardware compilation where a part of the functionality of the application is moved to an on-the-fly generated, hardware block, instantiated on an FPGA. Hot code fragments will be considered for this extra compilation stage. Section III-B discusses how we will select the suitable fragments in a communication-aware manner.

B. Optimal boundaries for partitioning

The MPSoC platform depicted in 1 is very well suited for splitting of computationally intensive kernels from the main processor. However, it is important to choose kernels with a high computation to communication ratio to reduce the extra communication overhead.

Our solution builds on the call graph of the application, in which all dynamically executed methods of the application are represented as nodes. The edges of the graph represent the method calls. By selecting sub trees of the call graph, as shown by dotted lines in Figure 3, we try to minimise communication. That is, each method A will be partitioned of the main tree together with all methods that are called by A. For example: if method 13 is a candidate for hardware acceleration, method 14 will also be implemented on the hardware accelerator. This approach is shown to be very successful in extracting suitable code fragments for off-loading [14].

C. Run-Time Partitioning and Re-Partitioning

For optimal communication-awareness, the Java program needs to be on-the-fly partitioned along the partition boundaries described in Section III-B. This can easily be done by adapting the JVM to count not only the execution frequency of each Java method, but to also count this for sub trees of the callgraph: that is, to accumulate the results for methods which are called, with the results of the calling method. This relatively small adaptation of the profiling phase in the JIT compiler suffices for the JVM to support run-time partitioning.

Another important question is: how can we dynamically move parts of the functionality of a running application to an accelerator? How does this hardware compilation work? The JVM supports three different situations: (i) the hardware configuration is loaded from a library [5], (ii) a generic hardware model is loaded from a library and specialised for a given set of parameters [6] or the hardware is generated on-the-fly [2][11].

D. Dynamic data allocation

This dynamic code partitioning was combined with the strategies for data allocation that we proposed in previous work [3]. This section gives a brief summary of the considered approaches.

Baseline—This algorithm allocates all objects in main memory close to the general-purpose processor. All memory accesses performed by any of the accelerators will be remote accesses which go over the relatively slow bus.

Optimal placement—Based on the joint usage pattern for all objects and measured during a complete run of an application, the optimal memory can be determined. We consider this as an optimal implementation within the given constraints and use it to compare all the other strategies.

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.

Self-learning allocation—In this strategy, the JVM decides at runtime where to allocate objects based on the usage patterns of previous objects. This is particularly useful in the dynamic environment of our Java based MPSoC platform, which decides at runtime whether to execute functionality on the general-purpose processor or on specific accelerators. The JVM continuously counts all memory accesses from both the main processor and the accelerators to each object in all memories. This can for instance be done through (sampled) instrumentation or hardware assisted profiling. Each object (or object group) has its own set of counters, one for the processor and one for each of the accelerators. At each point in time, comparing the counters will tell the JVM which component has accessed these objects the most up to now. New objects will be allocated in the memory closest to the component with the highest number of accesses.

IV. Experimental Results

For the evaluation of our approach, we use the DaCapo benchmark suite [4]. In an initial profiling run, we have determined for each benchmark the ten hottest methods, i.e. those accounting for the largest execution time. These methods and the methods called by these methods (the sub tree)

are considered, in our simulation, to be selected by the JIT compiler for hardware acceleration.

The MPSoC platform in this simulation consisted of two processing elements: a general-purpose processor running the JVM and one reconfigurable hardware accelerator which was pre configured for to be functionally equivalent with the hottest methods of the benchmark.

We have compared the remote access ratio for all benchmarks and for the four strategies described in Section . The remote access ratio is the relative number of memory transactions (read or write) that involve the global communication network, which is relatively slow. Therefore, the lower the remote access ratio, the better performance we get. From Figure 4 we learn that in this setup the self-learning strategy performs best for all benchmarks except for two benchmarks (`bloat` and `xalan`) that benefit more from the local allocation strategy.

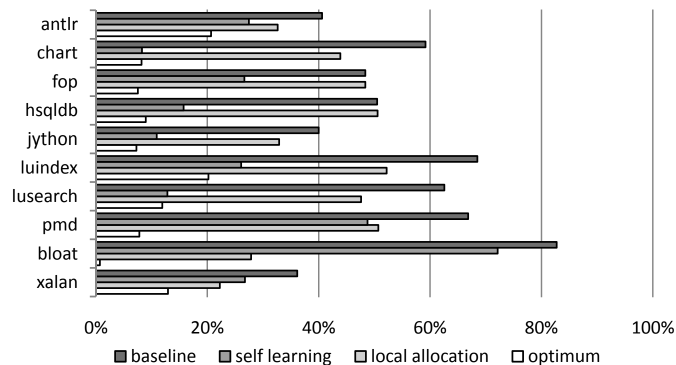

Fig. 4. Evaluation of several algorithms for dynamic memory allocation on an MPSoC platform with two heterogeneous cores and randomly mapped functionality.

V. Conclusions

We proposed an approach for adding extra flexibility to the design of embedded software on an MPSoC with the key concept of extending JIT compilation of the JVM to the dynamic off-loading of functionality from a general-purpose processor to a dedicated accelerator (DSP, FPGA, hardware block). We have also shown that this must be done carefully with respect to the communication overhead. And we conclude that, for a simple simulation, the extra overhead due to the distribution of the application over several processing elements on the MPSoC can be reduced drastically by means of the optimal memory allocation.

VI. Acknowledgments

Peter Bertels was supported by a PhD grant of 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).

References

[1] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003.

[2] Antonio Carlos S. Beck and Luigi Carro. Dynamic reconfiguration with binary translation: breaking the ILP barrier with software compatibility. In *Proceedings of the 42nd annual Design Automation Conference (DAC)*, pages 732–737, New York, NY, USA, 2005. ACM.

[3] Peter Bertels, Wim Heirman, Erik D'Hollander, and Dirk Stroobandt. Efficient memory management for hardware accelerated java virtual machines. *ACM Transactions on Design Automation of Electronic Systems*, 14(4):18, August 2009.

[4] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[5] Andrew Borg, Rui Gao, and Neil Audsley. A co-design 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)*, pages 58–67, New York, NY, USA, 2006. ACM.

[6] Karel Bruneel and Dirk Stroobandt. Automatic generation of run-time parameterizable configurations. In U. Kebschull, M. Platzner, and Teich J., editors, *Proceedings of the International Conference on Field Programmable Logic and Applications*, pages 361–366, Heidelberg, 9 2008. Kirchhoff Institute for Physics.

[7] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *POPL 1984: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302, New York, NY, USA, 1984. ACM.

[8] Philippe Faes, Peter Bertels, Jan Van Campenhout, and Dirk Stroobandt. Using method interception for hardware/software co-development. *Springer Design Automation for Embedded Systems*, pages 1–21, 2009.

[9] Philippe Faes, Mark Christiaens, Dries Buytaert, and Dirk Stroobandt. FPGA-aware garbage collection in Java. In *Proceedings of the international conference on Field Programmable Logic and Applications (FPL)*, pages 675–680, Tampere, Finland, 1 2005. IEEE.

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

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

[12] Roman Lysecky, Frank Vahid, and Sheldon X.-D. Tan. Dynamic fpga routing for just-in-time fpga compilation. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 954–959, New York, NY, USA, 2004. ACM.

[13] Elena Moscu Panainte, Koen Bertels, and Stamatis Vassiliadis. The MOLEN compiler for reconfigurable processors. *Transactions on Embedded Computing Systems*, 6(1):6, 2007.

[14] Sean Rul, Hans Vandierendonck, and Koen De Bosschere. Towards automatic program partitioning. In *CF 2009: Proceedings of the 6th ACM conference on Computing frontiers*, pages 89–98, New York, NY, USA, 2009. ACM.

[15] Michele Tartara, Simone Campanoni, Giovanni Agosta, and Stefano Crespi Reghizzi. Just-in-time compilation on arm processors. In *ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 70–73, New York, NY, USA, 2009. ACM.