FPGA-Aware Garbage Collection in Java

Philippe Faes, Mark Christiaens, Dries Buytaert, Dirk Stroobandt
Department of Electronics and Information Systems
Ghent University
Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium
email: {pfaes,mchristi,dbuytaer,dstr}@elis.ugent.be

Abstract— **During codesign of a system, one still runs into the impedance mismatch between the software and hardware worlds. This paper identifies the different levels of abstraction of hardware and software as a major culprit of this mismatch. For example, when programming in high-level object-oriented languages like Java, one has disposal of objects, methods, memory management, ... that facilitates development but these have to be largely abandoned when moving the same functionality into hardware.**

As a solution, this paper presents a virtual machine, based on the Jikes Research Virtual Machine, that is able to bridge the gap by providing the same capabilities to hardware components as to software components. This seamless integration is achieved by introducing an architecture and protocol that allow reconfigurable hardware and software to communicate with each other in a transparent manner i.e. no component of the design needs to be aware whether other components are implemented in hardware or in software.

Further, in this paper we present a novel technique that allows reconfigurable hardware to manage dynamically allocated memory. This is achieved by allowing the hardware to hold references to objects and by modifying the garbage collector of the virtual machine to be aware of these references in hardware. We present benchmark results that show, for four different, well-known garbage collectors and for a wide range of applications, that a hardware-aware garbage collector results in a marginal overhead and is therefore a worthwhile addition to the developer's toolbox.

I. Introduction

Hardware/Software codesign has become an indispensable tool in the designer's toolbox for building advanced embedded systems. It provides the designer with the best of both worlds: software (SW) development lends itself very well to building components that have a complex control-flow while hardware (HW) development excels at exploiting the maximum amount of parallelism that is present in the application and therefore achieves great speed-ups.

There is a catch: the SW and HW worlds speak totally different languages. A SW developer thinks in terms of data structures (preferably even object-oriented) and methods that operate on them. These methods are constructed using control-flow constructs such as loops, tests, method-calls, ... The SW developer's temporal view of the execution is often that of a purely sequential execution or maybe an execution with a mild amount of parallelism (using threads or message passing systems).

A HW developer on the other hand works with a model of the application that is much more physically oriented. Data resides in memories and registers and is operated upon by combinational logic built up from logic gates. Execution is often extremely parallel and is modelled by hundreds of parallel processes that function simultaneously.

All this results in a serious impedance mismatch when trying to unite the two worlds through codesign. A number of attempts to address these issues have already been made [1], [2], [3]. We focus on the interface between SW and *reconfigurable* HW because of reduced cost and ease of prototyping and because reconfigurable HW allows for time-multiplexing [4].

One cannot fail to notice that existing codesign support is still in its infancy so in [5] we proposed a codesign methodology with full support for object-oriented design and which bridges the gap between HW and SW in a much more elegant way. We presented a Java [6] runtime environment where complete transparency between a HW or SW implementation of tasks is aimed for i.e. where HW tasks have essentially the same capability to call methods, create objects, ... as their SW counterparts and can therefore readily be interchanged.[1] In addition it provides a shared-memory model (as opposed to message passing [7], [8] or remote procedure calls with marshalling [9] used in existing methodologies) where parallel executing tasks are modelled as threads communicating through read, write and synchronisation instructions on a shared memory.

Until recently our methodology only supported HW components which do not hold references to objects i.e. which do not have any impact on dynamic memory allocation. All dynamic memory management was performed by the SW acting as a proxy on behalf of the HW.

In this paper we present novel work enabling a HW component to actually create *and hold references to* dynamically allocated objects. Since HW components can now hold objects, the Java garbage collector needs to be made aware of the existence of these object references in order to find all objects that are still in use. In addition, most Java environments will perform a compaction phase after the garbage collection phase in order to create a contiguous area of free memory. To support compaction we enable the garbage collector to actually update references to objects residing in HW. Our techniques have been validated for four different garbage collectors and have been tested on benchmarks from the DaCapo Benchmark Suite [10].

The rest of this paper is organized as follows. First we explain the rationale of passing object references to HW. In Section 3 we describe the architecture and the protocol we

[1]The decision between HW and SW is made by the runtime environment. E.g. when the HW is executing one invocation of a method, another invocation can be executed in software.