Combining Processor Virtualization and Split Compilation for Heterogeneous Multicore Embedded Systems

Erven Rohou

INRIA
Campus de Beaulieu, 35042 Rennes CEDEX, France
erven.rohou@inria.fr

Abstract. Complex embedded systems have always been heterogeneous multicore systems. Because of the tight constraints on power, performance and cost, this situation is not likely to change any time soon. As a result, the software environments required to program those systems have become very complex too.
We propose to apply instruction set virtualization and just-in-time compilation techniques to program heterogeneous multicore embedded systems, with several additional requirements:

- the environment must be able to compile legacy C/C++ code to a target independent intermediate representation;
- the just-in-time (JIT) compiler must generate high performance code;
- the technology must be able to program the whole system, not just the host processor.

Advantages that derive from such an environment include, among others, much simpler software engineering, reduced maintenance costs, reduced legacy code problems... It also goes beyond mere binary compatibility by providing a better exploitation of the hardware platform.
We also propose to combine processor virtualization with split compilation to improve the performance of the JIT compiler. Taking advantage of the two-step compilation process, we want to make it possible to run very aggressive optimizations online, even on a very constraint system.

Keywords. heterogeneous multicore, virtualization, compilation, bytecode, annotations

1 Introduction

Complex embedded systems must provide a wide range of dedicated and demanding functionalities, such as communication, multimedia and user interface. Given their tight area and power constraints, it is impossible to provide those functions using homogeneous programmable architectures. Rather, they are composed of different subsystems, typically a host microcontroller running the system software (OS, user interface, peripheral drivers), a number of heterogeneous dedicated processors, such as DSP or VLIW, and dedicated hardware.

Dagstuhl Seminar Proceedings 08441
Emerging Uses and Paradigms for Dynamic Binary Translation
http://drops.dagstuhl.de/opus/volltexte/2009/1887

In the recent years, multicore processors have become mainstream also for general-purpose computing, exposing to a larger audience the need to better handle multiprocessing at the software level. Several roadmap documents [1,2,3] predict that, by the end of the next decade, computing systems — both general purpose and embedded — will feature many hundreds of cores.

This research project proposes to extend the domain of application of processor *virtualization* and to combine it with *split compilation* (also known as *multi staged compilation*) in the context of heterogeneous multicore systems. The anticipated goals are:

- to greatly reduce the burden put on software developers because of the maintenance of numerous compilers and tools needed to support various platforms (different models of cell phones, DVD players...), and to give them a simplified homogeneous view of their target systems;
- to improve the applicability of processor virtualization and the performance of programs running on top of such a layer, by making virtualization possible from the C (or C++) language;
- to open embedded systems to third-party developers, and to enable independent development of high performance application running not only on the host processor, but also on the more powerful media processors;
- to make aggressive runtime optimizations possible even for embedded systems thanks to split compilation;
- to propose a solution to the problem of application sustainability, taking advantage of virtualization to let applications survive architectures and to exploit new hardware features or additional degrees of parallelism, and at the same time, to free architects from the constraints of binary compatibility and legacy code.

The following section reviews the state of the art related to both virtualization and split compilation. Section 3 then details our proposal. We conclude in Section 4.

2 State of the Art – Related Work

2.1 Virtualization

Processor virtualization is not a new idea. It was already well known when it has been made popular for the PC market by Java [4] two decades ago, and today dedicated versions (Java Micro Edition) specifically target the embbeded domain.

CLI (Common Language Infrastructure) is the latest widespread processor independent format. Initially introduced by Microsoft under the name .NET, it is now an international standard [5,6]. CLI is multi language, it also support *managed* as well as *unmanaged* code. In other words it can be used for high level programmation paradigms (object orientation, garbage collection...) but it can also express the typical low-level programming style of the C language and it can

achieve better performance than Java. There are open source initiatives: Mono [7], Portable.NET [8] as well as commercial offers: the MicroFramework [9] for the embedded world from Microsoft.

Even though the processor virtualization technology is quite mature and popular, nowadays Java and CLI are applied only to the host processor of an embedded system. Still, we have shown that CLI makes a compact program representation [10], and that the bytecode can be efficiently compiled to native code [11]. Our research has been implemented in the open source compiler GCC [12] and contributed back to the community [13].

2.2 Split compilation

Split (or multi staged) compilation refers to the fact that the compilation process may be decomposed into several steps. Some standard static compilers already implement this strategy to some degree. For example, the GCC compiler [12] has two internal representations: the first one (GIMPLE) is a high level, target-independent representation, whose abstractions are much closer to the semantics of the source program; the second one (RTL) is a much lower level representation closer to the processor instruction set.

This definition of split compilation easily extends to all steps of a program "lifetime": static compilation, linking, installation, loading, runtime, and even the time between different runs (sometimes refered to as *idle time*). Each step brings additional knowledge about the runtime environment (shared libraries, operating system, processor, input values), opening the door to new classes of optimizations. Link time optimizers have the visibility of the whole application and apply interprocedural analyses and transformations (for example alto [14], or Diablo [15]). Runtime optimizations are applied by JIT compilers [16] and dynamic binary rewriters [17,18]. Idle time optimizers reoptimize an existing binary, taking into account profiling information produced by previous runs of the application (for example, Morph [19]).

LLVM [20] is an approach that aims to make lifelong program analysis and transformation available for arbitrary software, and in a manner that is transparent to programmers.

It is also a known fact that static optimizations can be too aggressive, to the point that some dynamic optimizers undo at runtime the transformations applied by the static compiler [21].

During the transformation from high level language, optimizers gain increasing knowledge about the final runtime environment, but they also gradually lose semantical information (arrays become pointers, typed integers become 32-bit amorphous values, etc.) There have been many attempts to help compilers with information it cannot derive by itself (C pragmas, HPF directives, OpenMP, DyC [22], aC# [23]), or to propagate information while lowering the program representation in order to make runtime optimizations more efficient (annotations in Java class file for register allocation [24,25], array bound checks removal [26], side effect analysis [27], etc. [28]).

Our own contribution in this field consisted in investigating the addition of annotation to the CLI file format for the benefit of a potential runtime vectorizer [29].

3 Proposal

Embedded systems have always been heterogeneous multicores. The hard constraints in terms of cost, area, performance, power consumption and real-time simply make a general purpose processor unfit for the task. The restricted application domain they serve makes symmetric multiprocessing very inefficient. With the exception of the host microcontroller, most hardware components used to be dedicated accelerators. However, the fixed cost of the design of a chip in the most advanced technologies now plays in favor of more programmable systems.

In today's PC market, multicore computers are sold for less than 400 euros at local supermarkets. Since technological and economical reasons have put a stop to the increase of clock frequency, performance improvements now come in the form of additional parallelism. It is expected that the number of available cores will grow to reach several orders of magnitude [1,2,3] by the end of the next decade.

Applications have a much longer lifetime than hardware. Most of them have been written with a sequential model in mind, or at best for a limited amount of parallelism. And while the number of available cores on a system is approaching hundreds, maybe thousands in the not-so-far future, it is unconceivable to rewrite millions of lines of code at each generation to exploit them. It will be increasingly important to be able to run applications on architectures with a higher degree of parallelism than they were designed for, and with different kinds of processors.

This document proposes to extend the split compilation approach and to combine it with processor virtualization to improve on a number of issues related to programming heterogeneous multicore systems.

Development Tools Developers need a complete toolchain for each kind of core present on their systems. Each compiler might (and often does) come from a different vendor, and thus it might be based on its own technology, controlled by its own set of command-line arguments, and expose specific optimization behavior to the programmer. Maintaining these tools, while not of theoretical content, is nevertheless a significant burden (and cost!) to the developers. Virtualization can contribute to the simplification by exposing a single environment to the developers and by postponing the specialization of the tools to the systems (*e.g.* to a JIT).

Debuggability, reproducibility Because the same application needs to run on different pieces of hardware, the source code tends to contain many conditional preprocessing directives (`#ifdef` in C). And because performance is critical, programmers rely on compiler intrinsics and *adhoc* command line flags to drive the compiler. This severely impacts code readability and thus productivity. Similarly, the application binary tested and debugged by the

programmer on his workstation is different from the one that eventually runs on the system. Virtualization makes it possible to run the *same binary* program on many hardware variants, including the developer's workstation.

Platform Openness Independent software vendors rarely have access to the tools to program the most powerful parts of the system, namely the DSP and the media processors. They are given access only the host processor, typically through a Java virtual machine and, most likely, a Java bytecode interpreter. Virtualization can make the whole platform programmable, and it opens the door to third-party high performance applications.

Binary Compatibility and Application Sustainability Hardware's lifecycle is much shorter than software's. Successful hardware vendors often have to make newer versions of their processors backward compatible, at a high cost. Virtualization lets them introduce radically new and efficient architectures, without having to worry about legacy code. But virtualization can even go beyond mere compatibility. Since the native code is generated on the system, old applications can exploit new hardware features that did not even exist when the application was first compiled.

Java applications have the reputation to be slow. This could be because the first virtual machines only had an interpreter, or a simple JIT compiler. As a consequence, all bytecode formats are often *a priori* perceived as inappropriate for performance intensive applications and for embedded systems. However, we have shown that a bytecode format produces a code size similar to dense instruction sets [10]. Moreover, we also proved that the bytecode representation can be compiled to native code with performance similar to static compilation [11].

As of today, virtualization is applied only to the host processor. In order to develop the above mentioned directions, we propose to extend it along three main directions:

- for languages such as C and C++;
- for high performance;
- for whole system programming, *i.e.* not only host processor, but also DSP, media processors and accelerators, grid of computing nodes, etc.

Processor virtualization can be of a great help to program heterogeneous multicore systems. Since the final code generation occurs at run time, mapping and scheduling of computations can be performed across all available processing nodes, independently from their underlying architectures.

Processor virtualization provides the framework for split compilation. The application is compiled in two stages: firstly, from source code to the intermediate format; secondly, from the intermediate format to native code. The latter occurs on the system and it is CPU and memory bound, but the former occurs on the developer's powerful workstation. Another significant difference is that the first one does not know the final target the code will run on, while the second has full knowledge. Dynamic optimization is not a replacement for static optimization, but a complementary optimization opportunity that leverages information not available until runtime.

We propose to take advantage of this two-step situation to transfer the complexity of compiler optimizations as much as possible to the first stage. When an optimization cannot be applied — either because it is target-dependent, or because it may increase code size too much, or because it is too costly to be applied at runtime — it might still be considered: an analysis can be run and its results encoded into annotations embedded in the intermediate format. The second stage can rely on the annotations and skip expensive analysis to implement straightforward code transformations. Annotations may also express the hardware requirements or characteristics of a piece of code (I/O required, benefits from hardware floating point support, etc.)

This multi stage process makes it possible to apply to embedded systems the most recent aggressive techniques like iterative compilation [30] or transformation in the polyhedral model [31].

Many complex optimizations, especially those considered too complex to fit a JIT compiler, can be revisited in the light of split compilation.

4 Conclusion

Recent technology trends started changing the way advanced architectures will look like in the future. The clock frequency has reached its limit, and additional performance will now come from an increased number of cores on the processor. This phenomenon drastically changes the way applications must be handled. Multicores will be ubiquitous. Applications will have to exploit an increasing degree of parallelism, made available in many possible ways: accelerators, grids of computing nodes, DSP...

Processor virtualization is a natural way to address heterogeneity. However, JIT compilers are constrained by the amount of memory and CPU time they can consume. In this paper, we propose to combine processor virtualization with split compilation techniques to overcome those limitations. An offline compiler can afford very aggressive analyses to collect relevant information about the application or the expected benefit of potential optimizations. The JIT compiler can rely on the precomputed information and combine it with the additional runtime knowledge to apply powerful transformations.

The ability to handle legacy code will be of utmost importance and processor virtualization and just-in-time (JIT) compilation combined with split compilation appears to be a reasonable way to efficiently handle the diversity of available computing resources in the near future.

References

1. De Bosschere, K., Luk, W., Martorell, X., Navarro, N., O'Boyle, M., Pnevmatikatos, D., Ramirez, A., Sainrat, P., Seznec, A., Stenström, P., Temam, O. In: High-Performance Embedded Architecture and Compilation Roadmap. Volume 4050/2007 of Lecture Notes in Computing Science. Springer Berlin / Heidelberg (2007) 5–29

2. Asanović, K., Bodik, R., Catanzaro, B., Gebis, J., Husbands, P., Keutzer, K., Patterson, D., Plishker, W., Shalf, J., Williams, S., Yelik, K.: The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California at Berkeley (2006)
3. Computing Systems Consultation Meeting: Research Challenges for Computing Systems – ICT Workprogramme 2009–2010. European Commission – Information Society and Media, Braga, Portugal (2007)
4. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. 2nd edn. Addison-Wesley (1999)
5. ECMA International Rue du Rhône 114, 1204 Geneva, Switzerland: Common Language Infrastructure (CLI) Partitions I to IV. 4th edn. (2006)
6. International Organization for Standardization and International Electrotechnical Commission: International Standard ISO/IEC 23271:2006 – Common Language Infrastructure (CLI), Partitions I to VI. 2nd edn. (2006)
7. Novell: The Mono Project. (`http://www.mono-project.com`)
8. Southern Storm Software, Pty Ltd: DotGNU project. (`http://dotgnu.org`)
9. Microsoft: Introducing the .NET Micro Framework. Product Positioning and Technology White Paper (2007)
10. Costa, R., Rohou, E.: Comparing the Size of .NET Applications with Native Code. In: Proc. of the 3rd International Conference on Hardware/Software Codesign and System Synthesis, Jersey City, NJ, USA (2005) 99–104
11. Cornero, M., Costa, R., Fernández Pascual, R., Ornstein, A.C., Rohou, E.: An Experimental Environment Validating the Suitability of CLI as an Effective Deployment Format for Embedded Systems. In: HiPEAC'08. Volume 4917 of Lecture Notes in Computer Science., Göteborg, Sweden (2008) 130–144
12. Free Software Foundation: The GNU Compiler Collection. (`http://gcc.gnu.org`)
13. Costa, R., Ornstein, A., Rohou, E.: CLI Back-End in GCC. In: GCC Developers' Summit, Ottawa, Canada (2007) 111–116
14. Muth, R., Debray, S., Watterson, S., De Bosschere, K.: alto: a link-time optimizer for the Compaq Alpha. Software: Practice and Experience **31** (2001) 67–101
15. Van Put, L., Chanet, D., De Bus, B., De Sutter, B., De Bosschere, K.: Diablo: a reliable, retargetable and extensible link-time rewriting framework. In: Proc. of the International Symposium On Signal Processing And Information Technology, Athens, Greece (2005) 7–12
16. Paleczny, M., Vick, C., Click, C.: The Java HotSpot^{TM} Server Compiler. In: Proc. of the Java Virtual Machine Research and Technology Symposium, Monterey, CA, USA (2001)
17. Bala, V., Duesterwald, E., Banerjia, S.: Dynamo: A transparent dynamic optimization system. In: Proc. of the Conference on Programming Language Design and Implementation. (2000) 1–12
18. Desoli, G., Mateev, N., Duesterwald, E., Faraboschi, P., Fisher, J.: DELI: a new run-time control point. In: Proc. of 35th Annual International Symposium on Microarchitecture, Istanbul, Turkey (2002) 257–268
19. Zhang, X., Wang, Z., Gloy, N., Chen, J.B., Smith, M.D.: System support for automatic profiling and optimization. In: Proc. of the 16th Symposium on Operating System Principles, Saint-Malo, France (1997) 15–26
20. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proc. of the International Symposium on Code Generation and Optimization, Palo Alto, CA, USA (2004)

21. Hazelwood, K., Conte, T.M.: A lightweight algorithm for dynamic if-conversion during dynamic optimization. In: Proc. of the International Conference on Parallel Architectures and Compilation Techniques, Philadelphia, PA, USA (2000) 71–80
22. Grant, B., Mock, M., Philipose, M., Chambers, C., Eggers, S.J.: DyC: An Expressive Annotation-Directed Dynamic Compiler for C. Technical Report UW-CSE-97-03-03, Univ. of Washington (1999)
23. Cazzola, W., Cisternino, A., Colombo, D.: [a]C#: C# with a Customizable Code Annotation Mechanism. In: Proc. of the 10th Symposium on Applied Computing, Santa Fe, NM, USA, ACM Press (2005) 1274–1278
24. Azevedo, A., Nicolau, A., Hummel, J.: Java annotation-aware just-in-time (AJIT) compilation system. In: Java Grande. (1999) 142–151
25. Jones, J.: Annotating Mobile Code for Performance. PhD thesis, University of Illinois at Urbana Champaign (2002)
26. Pominville, P., Qian, F., Vallée-Rai, R., Hendren, L., Verbrugge, C.: A framework for optimizing Java using attributes. In: Proc. of the 10th International Conference on Compiler Construction. Volume 2027 of Lecture Notes in Computer Science., Genova, Italy (2001) 334–354
27. Le, A., Lhoták, O., Hendren, L.: Using inter-procedural side-effect information in JIT optimizations. Technical Report 2004-5, McGill University - School of Computer Science (2004)
28. Krintz, C., Calder, B.: Using annotations to reduce dynamic optimization time. In: Proc. of the Conference on Programming Language Design and Implementation, Snowbird, UT, USA (2001) 156–167
29. Leśnicki, P., Cornero, M., Cohen, A., Fursin, G., Ornstein, A., Rohou, E.: Split compilation: an application to just-in-time vectorization. In: Workshop on GCC for Research in Embedded and Parallel Systems (GREPS'07), Braşov, Romania (2007)
30. Bodin, F., Kisuki, T., Knijnenburg, P., O'Boyle, M., Rohou, E.: Iterative compilation in a non-linear optimisation space. In: Workshop on Profile and Feedback-Directed Compilation, in Conjunction with PACT'98, Paris, France (1998)
31. Bastoul, C., Cohen, A., Girbal, S., Sharma, S., Temam, O.: Putting polyhedral loop transformations to work. In: Proc. of the 16th International Workshop on Languages and Compilers for Parallel Computing, College Station, TX, USA, Springer-Verlag (2003) 23–30