On the compilation of a parallel language targeting the self-adaptive virtual processor

Bernard, T.A.M.

Citation for published version (APA):
Summary

Computer-based systems are ubiquitous in daily life: in fridges which are regulated with an embedded Integrated Circuit (IC); in car control systems which manage the major driving features and engine controls; etc. The commodity computer market is driven by demands to deliver more efficient and reliable computing appliances at frequent time intervals: from a few months to a few years. These computing systems have to cope with the ever-growing increase of resource-intensive applications. IC (i.e. processor) improvements in computing systems are constrained to follow this never-ending quest for better performance. There are two major factors which contribute to this. The first improves process technology resulting in faster circuits by increasing a processor’s clock rate. The second is the increase in processing resources with integration of a larger number of transistors on chip. However, these improvements are nowadays reaching physical limitations: the more transistors present on a chip, the more physical issues arise resulting in excessive power consumption and problems with heat dissipation. Consequently, there is an urgent need to find other ways for microprocessor design improvements.

In the last few years, there has been a shift to parallel architectures in mainstream computing. Exploiting parallelism is not new; parallel programming paradigms, architectures and languages have been researched over the last three decades. This concurrency revolution is happening now in computing systems. Its impact on software systems appears at all levels, from the application level to the architecture level. Nonetheless, problems arise when dealing with concurrency. The main challenge is the efficient programming of such architectures: efficiency in building the applications from the developer’s perspective and efficiency in exploiting their full potential in terms of performance scalability. This fact remains, despite decades of research in parallel computing, because parallel programming is inherently difficult and non-intuitive for developers; humans tend to think sequentially. Unlike sequential programs, there are two issues in parallel application development. The first, and most common part, is the description of the algorithm required. This may be concur-
rent or may be automatically derived from a sequential description. The second, and by far the most difficult, is the management of concurrency. In this, the developer has to deal with issues of scheduling, mapping and dynamic resource management. In current approaches, the description and management of concurrency are not decoupled. This results in a mixture of concerns that overwhelm developers making parallel application development very difficult and error-prone. This lack of a clear separation of concerns also means a lack of appropriate high-level abstractions in both the architecture and application, which precludes portability between different platforms. Thus, what currently happens is that applications are either developed targeting specific platforms or existing applications are retargeted to a platform through a painstaking process of static application mapping and the introduction of platform-specific functionality into the application itself.

This concurrency revolution has an impact on software systems. Users utilize applications without knowledge of the machinery underneath. Applications are executed on the hardware; the operating system provides the interface between the software and the hardware. Having a multicore architecture requires an adapted toolchain in order to operate, comprising a concurrency-oriented operating system and corresponding applications. Users do not need to be aware of the machinery; however, developers use the adapted toolchain for software system development and must be aware of the machinery, at least the main concepts. Coupled with that, this toolchain must be adapted to handle the new features of this new concurrent target platform. Nowadays, the major issue of the computer community is to cope with the multicore programming menace. The problem is not that multicore architectures exist, but that the tools for exploiting them are not yet ready. In other words, the programmability of multicore architectures remains still a challenge. The main research focus of this thesis looks at a particular part of software system development which bridges the software side and the hardware side. Compilers are a major component in software system development. Often developers rely on them to optimize their code for an architecture about which they do not need to be completely aware. The role of the compiler is to take advantage of the targeted architecture with respect to the tasks to be accomplished within the programmed applications. This thesis, with an underlying compiler development for a new parallel programming language targeting a many-core architecture, looks at the impact of this concurrency revolution.

This thesis describes an abstract model called the Self-Adaptive Virtual Processor (SVP), which supports concurrent systems. Its main characteristic is that it provides a separation of concerns between concurrency description and concurrency management (scheduling and mapping). SVP provides abstractions that allow the description of an application’s concurrency at multiple levels of granularity, thus tackling one of the major issues with regards to parallel architectures and programming. The goal is to write an application capable of exploiting the full potential of any concurrent platform configuration. Concurrency management is delegated to the SVP implementation, which is directly
implemented in the processor’s ISA. Mapping is implemented in SVP by providing a mechanism to explicitly attach resources to components of a program and this is performed at execution time. Dynamic concurrency management has two main advantages. The first is that it allows applications to have a higher level of portability over different platform configurations (the number of processors differ). Secondly and most importantly, applications can take full advantage of any SVP platform without any platform-specific optimizations. SVP is flexible as it allows applications to exploit concurrency at all levels from the task level to the loop level and even at the instruction level if the implementation of SVP supports it. SVP captures data dependencies using dataflow scheduling between its component threads, which in turn introduces a mechanism by which SVP programs can be self-scheduled. This thesis introduces, as foundations, a hardware implementation of SVP at the level of a processor’s ISA and a software implementation with the extension of an imperative programming language.

This thesis investigates the changes and the challenges facing the integration of concurrency idioms and assumptions from a concurrent execution model into an existing sequential-based imperative-language compiler. The compilation schemes of the concurrent language implementation reuses and modifies the C-language standards. This thesis focusses on the differences of SVP compilation against conventional sequential compilation. Furthermore, this thesis investigates the impact of such an integration and reveals the feasibility of reusing existing antecedent compiler technology that has been proven to work. Nonetheless, such integration is a significant challenge and the side-effects are relevant to understand the quest of multicore programming tools.

At first glance, this work targets a specific audience of technical computer scientists involved in compilation. This work also represents a reflection on the limits of existing engineering methods when used to tackle the challenges of dealing with concurrency. The essence of the technical contribution of this work is then used as a theory of engineering on the limitations of current computing systems and other concurrency-based systems. In the context of facing the multicore programming menace, this work becomes relevant to an audience dealing with issues of concurrency-based compilation, language design, and multicore programming issues.