Sambamba: A Runtime System for Online Adaptive Parallelization

Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack

Saarland University, Saarbrücken, Germany
{streit,hammacher,zeller,hack}@cs.uni-saarland.de

Abstract. How can we exploit a microprocessor as efficiently as possible? The "classic" approach is *static optimization* at compile-time, optimizing a program for all possible uses. Further optimization can only be achieved by anticipating the actual *usage profile*: If we know, for instance, that two computations will be independent, we can run them in parallel. In the *Sambamba* project, we replace anticipation by *adaptation*. Our runtime system provides the infrastructure for implementing runtime adaptive and speculative transformations. We demonstrate our framework in the context of adaptive parallelization. We show the *fully automatic parallelization* of a small irregular C program in combination with our adaptive runtime system. The result is a parallel execution which *adapts to the availability of idle system resources*. In our example, this enables a 1.92 fold speedup on two cores while still preventing oversubscription of the system.

Keywords: program transformation, just-in-time compilation, adaptation, optimistic optimization, automatic parallelization.

1 Introduction

A central challenge of multi-core architectures is how to leverage their computing power for programs that were not built with parallelism in mind—that is, the vast majority of programs as we know them. Recent years have seen considerable efforts in automatic parallelization, mostly relying on *static program analysis* to identify sections amenable for parallel execution (often restricted to small code parts, such as nested loops). There also have been *speculative approaches* that execute certain code parts (identified by static analyses) in parallel and repair semantics-violating effects, if any.

While these efforts have shown impressive advances, we believe that they will face important scalability issues. The larger a program becomes, the harder it gets to precisely identify dependences between code parts statically, resulting in conservative approximations producing non-parallel and overly general code. The problem is that the actual environment and usage profile cannot be sufficiently anticipated [2]. Of course, one could resort to dynamic runtime techniques to determine dependences, but the initial overhead of dynamic analysis so far would not be offset by later performance gains. All of this changes, though, as soon as one moves the analysis and code generation from compile-time to runtime.

M. O'Boyle (Ed.): CC 2012, LNCS 7210, pp. 240–243, 2012.

Rather than analyzing and compiling a program just once for all anticipated runs, we can now reanalyze and recompile programs in specific contexts, as set by the input and the environment. Interestingly, it is the additional power of multi-core architectures that makes such a continuous adaptation possible: While one core runs the (still sequential) programs, the other cores can be used for monitoring, learning, optimization, and speculation. Moving from anticipation to adaptation enables a number of software optimizations that are only possible in such dynamic settings. First and foremost comes adaptive parallelization—that is, the execution of the program in parallel depending on the current environment.

2 The *Sambamba* Framework

The *Sambamba*[1] project aims to provide a reusable and extendable framework for online adaptive program optimization with a special focus on parallelization. With *Sambamba*, one will be able to introduce run-time adaptive parallelization to *existing large-scale applications* simply by recompiling; no annotation or other human input is required. Indeed, we aim to make parallelization an optimization as transparent and ubiquitous as, say, constant propagation or loop unrolling.

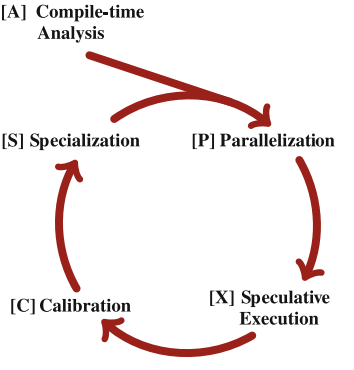

Fig. 1. *Sambamba* execution steps

Sambamba is based on the LLVM compiler framework and consists of a static (compiler) part and a runtime system. The framework is organized in a completely modular way and can easily be extended. Modules consist of two parts: *Compile-time parts* handle costly analyses such as inter-procedural points-to and shape analysis as used by our parallelization module. These results are fed into the *runtime parts*—analyses conducted at runtime which adapt the program to runtime conditions and program inputs. Obviously, it is crucial for the runtime analyses to be as lightweight as possible.

The flow of execution in the *Sambamba* framework is depicted in Figure 1:

[A] We use static whole-program **analyses** to examine the program for potential optimizations and propose a first set of parallelization and specialization candidates that are deemed beneficial. For long-running programs it might be a viable alternative to also run these analyses at runtime.

[P] The runtime system provides means for speculatively **parallelizing** parts of the program based on the initial static analysis and calibration information.

[X] We detect conflicts caused by speculative **executions** violating the program's sequential semantics and recover using a software transactional memory system [1] which we adapted to our special needs.

[1] *Sambamba* is Swahili for *parallel*, *simultaneously* or *side by side*.

[C] We gather information about the execution profile and misspeculations to **calibrate** future automatic optimization steps.

[S] Based on the calibration results, *Sambamba* supports generating different function variants that are **specialized** for specific environmental parameters and input profiles. These can then again be individually parallelized in the next step.

3 Adaptive Parallelization

3.1 Data Dependence Analysis

The main obstacle for parallel execution of program parts is data dependences over the heap. Parallel computation cannot start before all input data has been computed. In large irregular programs, the interprocedural data flow is hard to determine statically, so all known analyses only provide overapproximations.

In order to get a sound over-approximation of the existing data dependences, we use a state of the art context-sensitive alias analysis called *Data Structure Analysis* [3]. This information allows us to statically prove the absence of certain dependences.

3.2 Parallel CFG Construction

Given a regular control flow graph in SSA form, *Sambamba* creates the so-called parallel control flow graph (*ParCFG*). Unnecessary structural dependences defining an execution order are removed and replaced by real dependences caused by possible side effects.

We formulated the graph partitioning related problem as integer linear program (ILP). The solution of this ILP is then used to introduce so-called parallel sections (*ParSecs*). Each *ParSec* defines at least one fork point π_s and exactly one join point π_e for later parallel execution. Side-effect-free instructions might be duplicated in this step in order to facilitate parallelization.

We do not put special emphasis on loop parallelization and deal with general control flow instead. Very strong approaches of loop parallelization have been proposed and implemented during the last 30 years. Enriching some of these methods, like for example polyhedral loop optimization, with speculation is one of our ongoing projects.

3.3 Scheduling and Parallel Execution

In this step, *Sambamba* generates executable code from the *ParCFG*. This task includes the creation of an execution plan for concurrently executed parts as well as the generation of LLVM bitcode, which is translated into machine code by a just-in-time compiler.

In this demonstration, we only partition a region into parallel tasks if we could prove the absence of data dependences between them. Thus, the execution order of these tasks is not relevant. This will change as soon as we allow to speculate

on the absence of dependences. Then it may be beneficial to delay the execution of a task T until all tasks that T might depend on complete.

The assignment of tasks to processors is done dynamically by using a *global thread pool* initialized during load time of the program.

4 State of the Project

The demonstrated tool is a working prototype. Not every planned feature is fully implemented yet. Especially the features of the runtime-system are implemented on demand as we work on the modules for automatic parallelization.

At the time of writing, the following module independent parts are examples of implemented features:

- Method versioning and a general method dispatch mechanism
- A software transactional memory system supporting speculative execution
- Integration of the LLVM just-in-time compiler.

Concerning automatic parallelization, the demonstrated implementation is able to statically find sound candidates for parallelization. It identifies and rates data dependences which could not be statically proven to exist (may dependences) but prevent further parallelization. Execution adapts to the available system resources by dispatching between the sequential and a sound parallel version of parallelized methods.

For further details and news on the *Sambamba* framework please refer to the project webpage: *http://www.sambamba.org/*.

Acknowledgments. The work presented in this paper was performed in the context of the Software-Cluster project *EMERGENT* (www.software-cluster.org). It was funded by the German Federal Ministry of Education and Research (BMBF) under grant no. "01IC10S01". The authors assume responsibility for the content.

References

1. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP 2008), p. 237. ACM Press, New York (2008)
2. Hammacher, C., Streit, K., Hack, S., Zeller, A.: Profiling Java programs for parallelism. In: Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering, pp. 49–55. IEEE Computer Society (2009)
3. Lattner, C., Lenharth, A., Adve, V.: Making context-sensitive points-to analysis with heap cloning practical for the real world. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2007), pp. 278–289. ACM, New York (2007)