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. 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.

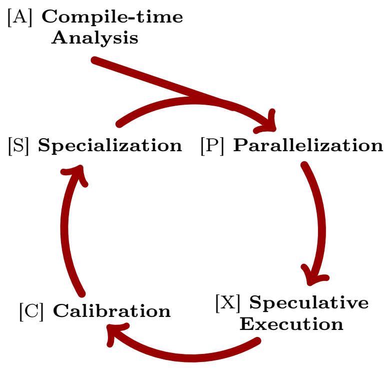


Fig. 1. *Sambamba* execution steps

Sambamba is based on the LLVM compiler framework [3] 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] *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* [4]. 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 use an integer linear programming (ILP) approach to graph partitioning to form 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 [1], 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. Grosser, T., Zheng, H., Aloor, R., Simbürger, A., Größlinger, A., Louis-Noël, P.: Polly—Polyhedral optimization in LLVM. In: First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011). pp. 1–6 (Mar 2011)
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., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. p. 75. IEEE Computer Society, Palo Alto, California (2004)
4. Lattner, C., Lenharth, A., Adve, V.: Making context-sensitive points-to analysis with heap cloning practical for the real world. SIGPLAN Not. 42(6) (2007)

A Description of the Planned Demonstration: Adaptive Parallel Execution

In our demonstration we will show the automatic parallelization and adaptive execution of a small irregular C program. The program source is given in Listing 1.1. Although a human developer may easily spot parallelization opportunities, the program contains two main challenges for automatic parallelization:

1. It does not contain any loops. Most parallelization approaches do only focus on parallelizing loops. This is reasonable since most often the biggest speedup can be educed out of loops. Nevertheless, this simple application can be accelerated by a factor of two which is missed by most existing approaches. Since *Sambamba* is not limited to loop parallelization, it is able to exploit this potential.
2. The more daunting challenge is the application's irregularity. It makes use of linked lists and iterates over them during execution. By making use of a state of the art interprocedural points-to analysis [4] *Sambamba* is able to deal with this irregularity.

A.1 Setup

Our demonstration is planned to consist of two parts: A short introduction of the sample application and what is going to be shown (two to three slides) followed by a live demonstration of *Sambamba* in comparison to gcc (see the next section). Afterwards an explanation is given of what is done in the background by *Sambamba*.

A.2 Execution

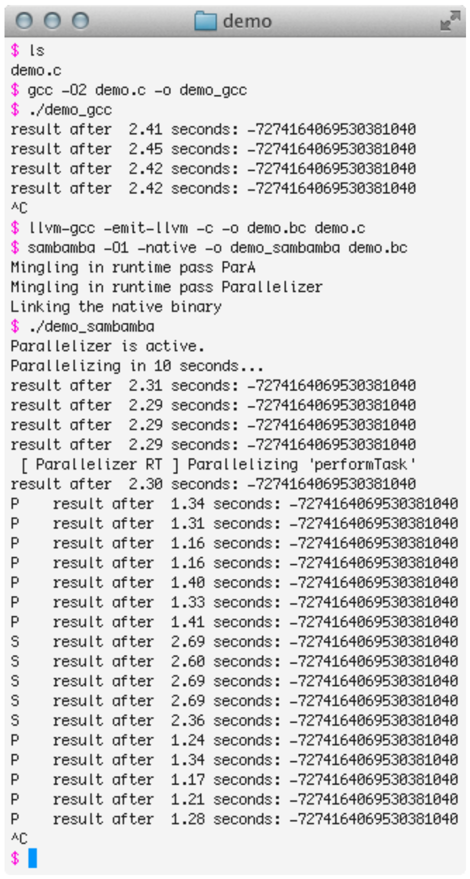
In order to demonstrate how to use *Sambamba* we prepared a short execution. The screenshot in Figure 2 shows all the steps which are described in more detail in the following sections.

Sequential Execution For reasons of comparability we first compile the sample application using a current version of gcc with optimizations enabled. The resulting binary is executed: The result of the computations can be seen, as well as the execution time of roughly 2.4 seconds.

Compilation Using *Sambamba* Afterwards, we compile the application using *Sambamba*: First, the *LLVM* bitcode is produced using llvm-gcc without any optimizations applied. Afterwards, *Sambamba* is invoked with the produced bitcode. Optimization is enabled and two *Sambamba* modules are mingled in: *ParA*, which is the **par**allelization **a**nalysis finding parallelization candidates and producing an execution plan. And the *Parallelizer* which is responsible for generating the parallel code. In the case of speculative execution, the *Parallelizer* additionally performs transactification of the code.

```c
typedef struct list {
  struct list *Next;
  int Data;
} list;

/*
 * Definitions for methods makeList, hashList and freeList
 * omitted. Please refer to http://www.sambamba.org/ for
 * the full sources.
 */

long performTask(int size) {
  list *X = makeList(size);
  list *Y = makeList(size);

  long hash_X = hashList(X);
  long hash_Y = hashList(Y);

  freeList(X);
  freeList(Y);

  return hash_X * hash_Y;
}

struct timeval start, end;

int main() {
  while (1) {
    gettimeofday(&start, 0);
    long res = performTask(1 << 10);
    gettimeofday(&end, 0);

    double secs = (end.tv_sec - start.tv_sec) +
                  1e-6 * (end.tv_usec - start.tv_usec);

    printf("result␣after␣%5.2f␣seconds:␣%ld\n", secs, res);
  }

  return 0;
}
```

Listing 1.1. Irregular sample application written in C.



```
$ ls
demo.c
$ gcc -O2 demo.c -o demo_gcc
$ ./demo_gcc
result after  2.41 seconds: -7274164069530381040
result after  2.45 seconds: -7274164069530381040
result after  2.42 seconds: -7274164069530381040
result after  2.42 seconds: -7274164069530381040
^C
$ llvm-gcc -emit-llvm -c -o demo.bc demo.c
$ sambamba -O1 -native -o demo_sambamba demo.bc
Mingling in runtime pass ParA
Mingling in runtime pass Parallelizer
Linking the native binary
$ ./demo_sambamba
Parallelizer is active.
Parallelizing in 10 seconds...
result after  2.31 seconds: -7274164069530381040
result after  2.29 seconds: -7274164069530381040
result after  2.29 seconds: -7274164069530381040
result after  2.29 seconds: -7274164069530381040
 [ Parallelizer RT ] Parallelizing 'performTask'
result after  2.30 seconds: -7274164069530381040
P    result after  1.34 seconds: -7274164069530381040
P    result after  1.31 seconds: -7274164069530381040
P    result after  1.16 seconds: -7274164069530381040
P    result after  1.16 seconds: -7274164069530381040
P    result after  1.40 seconds: -7274164069530381040
P    result after  1.33 seconds: -7274164069530381040
P    result after  1.41 seconds: -7274164069530381040
S    result after  2.69 seconds: -7274164069530381040
S    result after  2.60 seconds: -7274164069530381040
S    result after  2.69 seconds: -7274164069530381040
S    result after  2.69 seconds: -7274164069530381040
S    result after  2.36 seconds: -7274164069530381040
P    result after  1.24 seconds: -7274164069530381040
P    result after  1.34 seconds: -7274164069530381040
P    result after  1.17 seconds: -7274164069530381040
P    result after  1.21 seconds: -7274164069530381040
P    result after  1.28 seconds: -7274164069530381040
^C
$
```

Compiling using gcc for comparison
Executing the produced binary

Creating the LLVM bitcode using *llvm-gcc*
Linking the native binary using *Sambamba*

Executing the produced binary

Dynamically creating and registering a parallel version of method ‘performTask’

External compute intensive process started

External process finished

Fig. 2. Execution of a *Sambamba*-enabled program. First, the sample program (Listing 1.1) is compiled and executed using gcc. In the second step a binary is created and executed using *Sambamba*. The depicted execution was recorded on an Intel Core 2 Duo machine with a clock rate of 2.4 GHz.

Execution and Dynamic Parallelization The binary produced in the previous step is executed. For demonstration purposes parallelization is delayed and regular sequential execution starts. The same result as with the gcc compiled version and roughly the same execution time can be observed. After 10 seconds, the most profitable method, *performTask*, is parallelized and parallel execution immediately starts. Again for reasons of demonstration, the dispatcher mechanism is instrumented to print a P to console when choosing the parallel version of a method, and an S when choosing the sequential version. The execution time drops from 2.3 seconds to 1.2 seconds.

Adaptation to Available System Resources When running an external process which occupies at least one of the two available cores, the *Sambamba* runtime system falls back to sequential execution. This is demonstrated by running an arbitrary second process, in our example a simple endless loop. The fallback to sequential execution is a demonstration of one possible way of adaptability. Though fairly simple it is useful since it prevents oversubscription of the system. Most parallelization approaches parallelize in a greedy way which might not be the best solution under all circumstances.

A.3 Explanation

The demonstrated version of the parallelization module works on control flow graphs in SSA form and produces a so-called *ParCFG*. This is an extended version of a regular CFG which for example provides primitives for fork/join parallelism.

The control flow graph as given by the LLVM framework for the *performTask* method is shown and explained. A simplified version of this cfg for sequential execution is shown in Figure 3.

After a brief description of how the *ParA* module of *Sambamba* finds candidates for parallelization and produces execution plans using an ILP based approach, the produced *ParCFG* of the *performTask* method is shown. A simplified version abstracting away some technical details is given in Figure 4.

A.4 Ongoing Work

If time is left we will shortly describe our ongoing work. At the time of writing we are implementing two further modules for parallelization in *Sambamba*: One for speculation support for loop parallelization in the polytope model, and one for PDG based whole program parallelization using integer linear programming. The first approach will make heavy use of speculative execution. The latter one will support runtime adaptive parallelization and speculative execution.

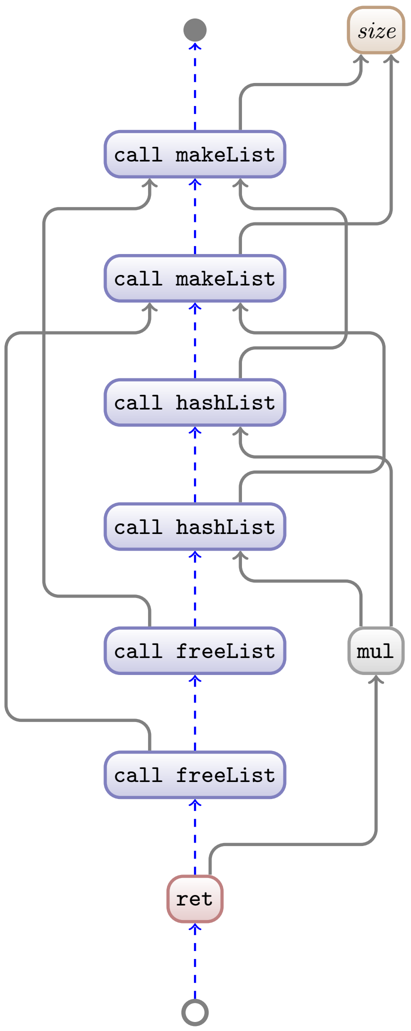


Fig. 3. Regular CFG of the *performTask* method. Dashed arrows depict structural dependences of instructions with possible side-effects.

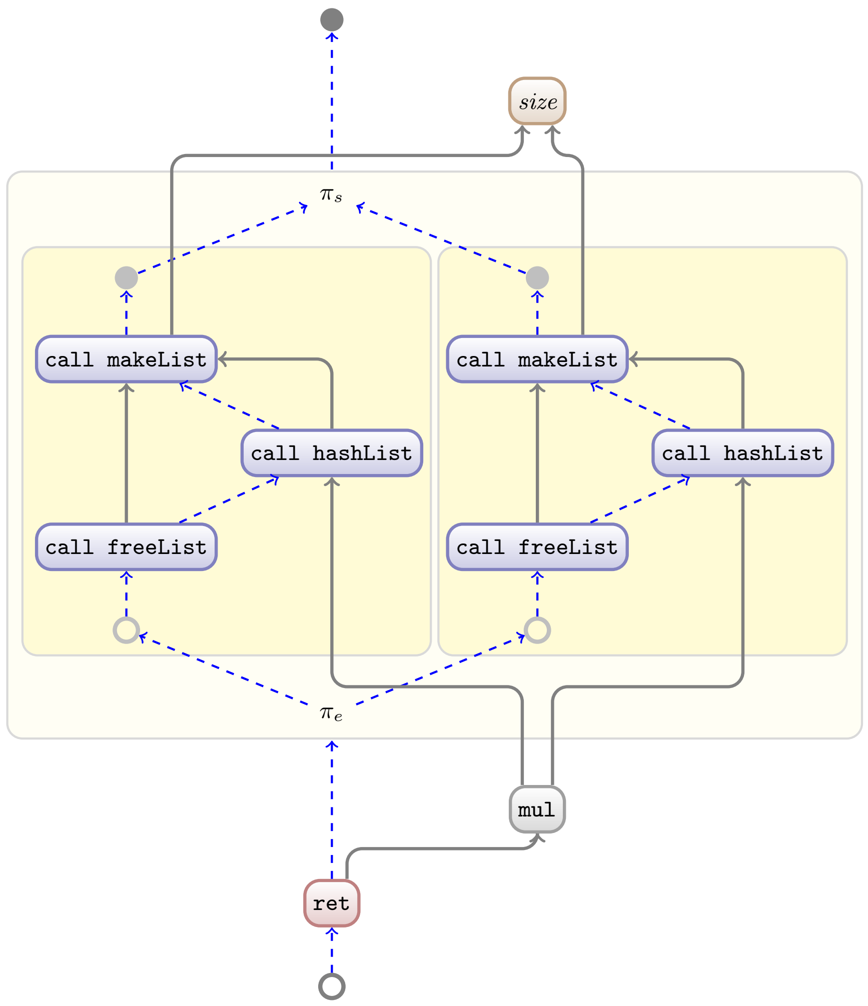


Fig. 4. ParCFG for parallel version P_0 of the *performTask* method as automatically derived by *Sambamba*. The outer box depicts a so-called parallel region consisting of transactions depicted by the inner boxes. Each parallel region is entered via at least one π_s nodes and left via the one π_e. A π_s forks parallel execution and π_e joins again after all contained transactions completed.