Sambamba: Runtime Adaptive Parallel Execution

Kevin Streit Clemens Hammacher
Saarbrücken Graduate School of Computer Science
Saarland University
Saarbrücken, Germany
{streit|hammacher}@cs.uni-saarland.de

Andreas Zeller Sebastian Hack
Saarland University
Saarbrücken, Germany
{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, conservatively optimizing a program while keeping all possible uses in mind. 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. However, brute force parallelization may slow down execution due to its large overhead. But as this depends on runtime features, such as structure and size of input data, parallel execution needs to *dynamically adapt* to the runtime situation at hand.

Our SAMBAMBA framework implements such a dynamic adaptation for regular sequential C programs through *adaptive dispatch* between sequential and parallel function instances. In an evaluation of 14 programs, we show that automatic parallelization in combination with adaptive dispatch can lead to speed-ups of up to 5.2 fold on a quad-core machine with hyperthreading. At this point, we rely on programmer annotations but will get rid of this requirement as the platform evolves to support efficient speculative optimizations.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation, Compilers, Optimization, Retargetable compilers, Run-time environments; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming

General Terms Parallelization, Performance

Keywords automatic parallelization, just-in-time compilation

1. Introduction

To exploit the power of modern multi-core architectures, one needs to compute in parallel — either by writing new parallel programs, or by parallelizing existing (sequential) programs. Both these pose grand challenges to research.

As an ongoing example, consider the *hashList()* function shown in Listing 1. It computes a hash value for a linked list by recursively combining the hash value of the first element with the hash of the remainder of the list. Assuming that *hashElem()* and *hashList()* have no side effects and do not interfere via accesses to the same memory regions, the two calls to *hashElem()* and *hashList()* could be run in parallel.

```
long hashList(list *x) {
    if (x == 0)
        return 0;

    return hashElem(x) +
        31 * hashList(x->next);
}
```

Listing 1. Irregular recursive sample method written in C.

The problems in automatically parallelizing this seemingly simple function statically are manifold though: To start with, the static code analysis of the compiler must be conservative. In order to parallelize this function the compiler has to statically prove that the *hashElem()* function has no observable side effects, such as raising a signal. Furthermore the *hashElem()* function is not allowed to write to global state or even to local state in the corresponding list element if the list cannot be proven to be non-circular. The compiler may not be able to determine if *hashElem()*, and thus *hashList()*, fulfills these criteria.

What a compiler could do, however, is to speculatively parallelize — that is, to execute the functions in parallel first, and to revert to sequential execution in case of conflicts. Detecting such conflicts and allowing for rollbacks induces overhead, however, and the compiler is not able to decide when such overhead may pay off; in particular for such a "small" function as *hashList()*. Due to this overhead, invoking *hashList()* on a "short" list and with few work to be done in *hashElem()* will best run sequentially, while on a "long" list or with costly *hashElem()* computations, it may better run in parallel. But where is the boundary between "short list" and "long list", between "few work" and "costly computation"? On which system? For which load?

Maybe we should leave the answer to such questions to the expert: With manual parallelization, the programmer provides hints or instructions on when and where to parallelize. In algorithms like parallel sorting, for example mergesort, we usually see code that switches to sequential execution when the length of the array to sort falls below a certain, constant, threshold. The assumption here is that spawning parallel threads or even tasks will not pay off below that threshold. But how can a programmer decide in the general case what threshold is optimal? Again, on which system?

In *hashList()* it is even worse: We cannot easily get the length of the linked list without traversing it. So how would an efficient check, deciding on when to switch to sequential execution look like?

When it comes to profitability of parallel execution, we have further problems: The potential overhead inherently depends on

ADAPT '13 January 22, 2013, Berlin, Germany.
Copyright © 2013 ACM 978-1-4503-2022-1/13/01. . . $15.00

the specific usage profile, which is not known at compile-time, and which may even vary during runtime. The resulting program — whether compiled through a specialized parallelizing compiler or produced manually by rewriting the code to run in parallel — will thus most likely be specialized towards a specific context, effectively breaking the promise of "write once, run anywhere".

We believe that program code should remain independent of the execution context; thus, parallelization should eventually be left to the compiler and a runtime environment. However, the large overhead of parallelization, which pays off only for specific inputs, calls for a runtime adaptive approach of parallel execution.

In particular, we must:

1. **Make programs adapt to their environment.** In terms of parallelization, there is no "one size fits all" solution. Instead, we must ensure that programs can choose between several instances of compiled methods: sequential or parallel, specialized or general, speculative or conservative, depending on the execution environment at hand.
2. **Have programs constantly optimize themselves.** The programmer should not be burdened with the choice of what and when to parallelize. Instead, we anticipate that programs learn during the execution which optimization strategies work best. The program would thus choose whether to assume a "short" or a "long" list, taking the current context into account.
3. **Move parallelization related decisions to run time.** For optimizations such as the ones above, we need traditional compile-time analyses, but conduct them at run time. This means that the separation of "compile time" and "runtime" becomes blurred; we would rather think of "constant recompilation" to fit the program best to the current situation. For instance, the compiler could produce specific optimized versions of *hashList()*— a sequential or speculatively parallel version if exceptions may be raised, and a non-speculative, highly efficient parallel version if it can decide that this is not the case.

It is clear that all these analyses and optimizations incur overhead; so how can we reasonably expect the program to run faster? While the program is not yet parallelized, we can make use of the idle cores to monitor the running program and to decide if and how to adapt. Once adaptation is done, and the input is stable, the extra analyses are no longer needed and are thus "turned off". In addition to pure performance, one could also think of optimizing for energy efficiency or memory consumption.

In this paper we apply an static fork-join parallelizer based on program dependence graphs (PDGs), which is built into the *Sambamba* framework [15], to find opportunities for parallelization. We use runtime adaptive code generation to schedule the found candidates based on execution time profiles collected at runtime. Finally we explore the influence of four different mechanisms to dynamically dispatch between the generated parallel and sequential code versions. On a sample application containing the *hashList()* function of Listing 1, we achieve a maximum speedup of **5.2 fold** as compared to sequential execution on a quad-core machine with hyperthreading.

This paper proceeds as follows: In the next section we talk about related work. In Section 3 we introduce the fork-join parallelization facilities of *Sambamba* and explain how code is adaptively generated. In Section 4 we elaborate on different methods of dispatching between sequential and parallel execution. In Section 5 we evaluate our approach on 14 programs taken from the suite of *cilk* example applications. Finally we conclude and describe our ongoing work in Section 6.

2. Related Work

Automatic parallelization, even at runtime, is an active research area and powerful approaches exist. However, most of the work known to the authors has a severe limitation that we circumvent in *Sambamba*: Only one version of each function is kept and optimized for the "general case". Execution time and branch profiles are either statically estimated or collected in special profiling runs or at runtime. Once enough information has been collected to identify so called hot regions — mostly loops — those are parallelized if appropriate. This is where purely static parallelization ends. Dynamic approaches keep monitoring the performance of parallelized code. In case it is deemed unbeneficial in terms of execution time, it is dropped and replaced by the sequential version for future executions. The problem with all these approaches is that the "general case", for which the code is optimized, has to be identified in terms of representative input data and execution environment. This is by far no easy task, in particular if input and environment change during execution. In this paper we show that the runtime overhead induced by continuously monitoring the running application actually pays off in several cases. The enabled continuous adaptation leads to increased performance, as compared to brute force or one-way parallelization.

Due to space limitations we stay very abstract here, but want to mention one recent approach that seems very interesting: The *Parcae* system by Raman et al. [10] provides a flexible parallel execution environment and promises to allow for holistical optimization of a parallel system instead of mere parameter tuning. The Parcae system is able to dynamically choose between two modes of loop parallelization: parallel stage decoupled software pipelining (*PS-DSWP*) [11] and *DOANY* [17]. In the case of PS-DSWP it is also able to dynamically tune the number of replicated parallel stages. The results reported in the above mentioned work support our intention that runtime adaptivity going beyond parameter tuning can dramatically improve the performance of existing parallelization schemes.

In contrast to Parcae, parallelization in *Sambamba* so far focuses on non-loop parallelization. Most of the existing work on automatic parallelization argues that the highest potential for parallel execution is hidden in loops, and they are certainly right. Nevertheless it has been shown [2, 9] that the available parallelism exceeds the execution of loops. We think that, in particular when leaving the well analyzable class of applications from a mathematical domain, parallelizing non-loop code becomes crucial.

3. Runtime Adaptive Parallelization

Our runtime adaptive fork-join parallelizer, *ParA*, is implemented as a module to the *Sambamba* framework. It consists of two parts: a compile-time component, performing possibly costly analyses and program transformations offline; and a runtime component, building on statically gathered information and continuously collected runtime profiles to perform online adaptive optimizations. Please refer to the project website [14] for details on *Sambamba*.

3.1 ParA: Static Component

Each parallel version of a sequential program has to respect control and data dependences in order to behave semantically equivalent. This property makes the program dependence graph (*PDG*) [4] a suitable representation for automatic parallelization. Its edges represent the control and data dependences of a function and efficient algorithms exist to schedule executable code from PDGs [13].

The static component of *ParA* uses the PDG as its internal representation to automatically find promising code locations, suitable for parallel execution.

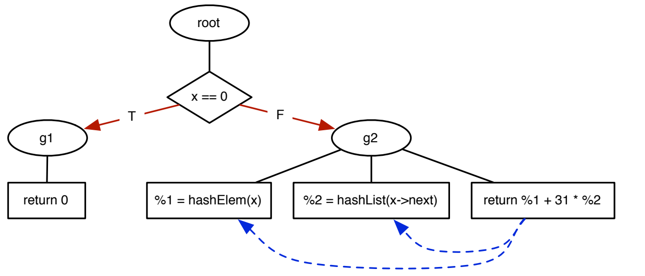

Figure 1. The simplified PDG of *hashList()*: data dependences are depicted by dashed arrows, control dependences by solid arrows.

Control dependence computation is conducted on the preprocessed *LLVM* [7] intermediate representation of each function, following Sarkar [12]. In order to compute the data dependences, an interprocedural and context-sensitive points-to analysis called data structure analysis (*DSA*) [8] is used. Each PDG node is annotated with a representation of its read and write effects to the abstract memory cells as reported by the DSA. Observable side effects such as system calls are modeled as read and/or write accesses to a special cell representing non-memory effects (*NME*).

As an example, the PDG of the *hashList()* function is shown in Figure 1. The PDG contains nodes of three different types:

- Regular nodes, depicted as boxes, represent simple instructions, or, as in our case, basic blocks.
- Decision nodes, such as conditional branches, switches or possibly exception throwing calls (invokes), are depicted as diamonds.
- Finally, group nodes, depicted as ovals, contain possibly multiple nodes sharing the same control conditions.

Each group node represents a control condition, which is the conjunction of all conditions of decision nodes on the path from the designated PDG *root* node to the corresponding group. Once this condition is fulfilled, all its child nodes are to be scheduled for execution. Only the data dependences between the subgraphs reachable from a group node restrict parallel execution. Within one group node, no complex control flow has to be taken into account; a property that makes the group nodes particularly interesting in the context of automated fork/join parallelization.

After computing the PDG, an ILP solver[1] is used to compute a parallel fork-join schedule for each group node in the PDG. The constraints of the linear program are formed by the data dependences between the subgraphs reachable from the group nodes, estimates of the execution times of called methods, branch profiles, which implicitly includes loop trip counts, and possible parallelization overhead.

The ILP formulation allows to speculate on the non-existence of dependences and takes ignored dependences into account by marking the parallel transactions containing the source and target nodes of the dependence correspondingly. At runtime, these transactions need protection by a speculation system, such as transactional memory (*TM*) [6]. The *Sambamba* runtime system provides speculation protection in the form of software transactional memory (*STM*) using an adapted version of TinySTM by Felber et al. [3]. If speculation is necessary, additional overhead is incurred and taken into account by the ILP. The optimization goal is to minimize the critical path execution time for each PDG group node.

[1] IBM Cplex: `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`

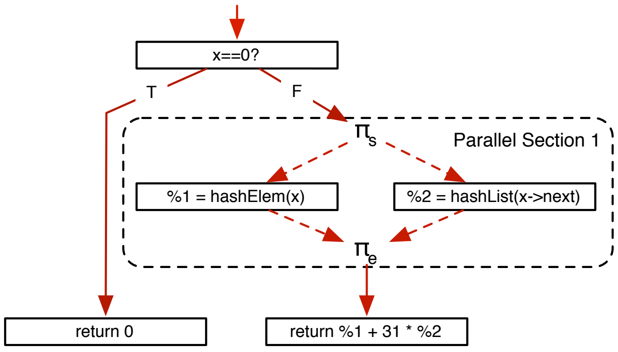

Figure 2. The simplified ParCFG of *hashList()*: The highlighted parallel section is entered via the π_s; parallel execution is synchronized at the corresponding π_e node.

Due to space limitations, we cannot go into details of the ILP formulation. Instead we refer the reader to our upcoming publications.

The need for runtime adaptive parallelization becomes particularly apparent if speculation is involved, as the possible overhead is large, in case misspeculation occurs too often. Unfortunately the probability of misspeculation is inherently input dependent and cannot be anticipated statically.

The local schedules, being compiled from the solution of the ILP per group node, are persisted as candidates for refinement and adaptive combination at runtime.

3.2 ParA: Runtime Component

The runtime component of *ParA* first reads the parallelization candidates found at compile time. These candidates are analyzed for possibly called methods which in turn are profiled for collecting their execution time. Taking the gathered information into account, the most promising local candidates for parallel execution are selected and combined. Parallel code is scheduled for the best combination. The result is a so called parallel control flow graph (*ParCFG*): a control flow graph annotated with fork and join points that mark the entrances and exits of parallel sections. Fork nodes are called π_s (π-Start) and join nodes π_e (π-End). The ParCFG resulting from the parallelization of *hashList()* is shown in Figure 2.

Since execution times might change depending on input, system load or parallelization, we keep profiling enabled for relevant methods. Upon a significant change in the execution time of profiled functions, scheduling decisions are reassessed and possibly revoked.

This is particularly important if speculation is involved: parallel sections that contain the source and target nodes of a speculatively ignored dependence are guarded by an extended STM system. One crucial extension that has been implemented in *Sambamba* is the incorporation of a commit order between speculative transactions. This commit order guarantees progress and allows to speculate on the non-existence of non-memory side effects, such as for example calls to printf, from within a speculative transaction. If such a call is encountered during execution, the corresponding transaction waits until all its commit predecessors are done; afterwards it verifies that no conflict occured so far and proceeds. In that situation, commit orders guarantee that the transaction will definitely be able to commit and no rollback can occur. It is thus safe to execute the externally observable function call.

The current speculation system of *Sambamba* comes in the form of an extended STM system. It was designed for very specific use cases and is not suitable for large transactions and recursive parallel execution. Although it works in that cases, the introduced overhead can by far exceed the benefit. To bridge the gap of a missing efficient speculation system where STM is inappropriate, we implemented a set of pragmas allowing to give hints on where to speculate and where not. These pragmas are only a temporary solution that allow to initialize the system to the state that is reached, once adaptive code generation has learned from frequent misspeculation. Note that these pragmas only affect the speculation system; parallelization stays fully automatic. No hints are given to the system on where to parallelize and where not. In Section 6.1 we shortly explain how we will supersede these speculation hints.

In the current implementation, we keep only one parallel version of each function in addition to the sequential version. This version is continuously adapted to the situation at hand.

Gathered runtime and profiling information is persisted by the *Sambamba* runtime system and needs not be recollected from scratch upon program termination and reexecution.

4. Runtime Adaptive Dispatch

In our earlier work [15] we showed how brute force parallelization of different methods can interfere, depending on the input. This interference can lead to dramatically decreased performance of the subject application; an effect that is effectively neutralized by the current implementation of *ParA*.

As described in the previous section, *ParA* keeps up to two versions for each method at runtime: the original, sequential version, and the currently best suited parallel version. If both versions exist, it is up to a dynamic dispatch mechanism to decide which version to choose upon a call to the corresponding function.

Based on our inspections of manually and observation of automatically parallelized code, we came up with four different lightweight dispatch schemes:

- **none**: No explicit dispatcher is used. The parallel version is always chosen if it exists. This dispatch scheme corresponds to brute force spawning of parallel tasks.
- **tif**: The dispatcher takes into account the number of tasks in flight. This is, the number of tasks that have been scheduled for parallel execution but not yet finished, i.e. the number of tasks waiting for execution plus the number of tasks currently executing. This dispatcher chooses the sequential version of the function if the number of tasks in flight is greater than four times the number of processors of the current system. Otherwise, the parallel version is chosen.
- **tnd**: This dispatch scheme takes into account the nesting depth of parallel tasks. It is particularly well suited for dispatching recursive functions like for example mergesort or a recursive implementation of the fibonacci function. Once the system is fully loaded, there is no use in further spawning parallel tasks. The sequential version of a function is chosen, if the task nesting depth exceeds the logarithm of the number of processors, the parallel version otherwise. Taking the logarithm base two favors parallel over sequential execution. If more than the minimum of two tasks are spawned at each nesting level, the system might already be fully loaded at a lower depth.
- **load**: The load dispatcher chooses the version to execute based on the current system load. If *Sambamba* detects that foreign tasks executed on the same system utilize a large amount of the available computing resources, no further parallel tasks are spawned. This dispatcher effectively prevents from oversubscription of the overall system.

Note that parallel code scheduling, as described in Section 3, is orthogonal to the adaptive dispatcher. If suitable, *ParA* provides the parallel version of a function independent of the chosen dispatcher. In the current implementation, a dispatch mechanism has to be selected when registering a new version of a function. In later incarnations, it will be up to the adaptive runtime system to automatically choose the dispatch mechanism that best fits the registered method. Our results corroborate the necessity to dispatch differently depending on the subject function in order to maximize performance.

5. Evaluation

In order to evaluate the success of *Sambamba*, *ParA* and the different adaptive dispatch schemes, we utilize the example suite of *cilk* [1]. It contains differently sized programs (55 to 3,265 LOC), annotated with *spawn* and *sync* primitives. All of the programs are recursive and as such particularly well suited for exploiting fork/join style parallelism as done by *Sambamba*. Moreover, parallelism scales with the number of used cores, provided the input size is big enough. In this section we evaluate parallel execution in *Sambamba* on systems with 4 and 8 cores respectively.

Three of the used cilk programs (*fft*, *magic* and *plu*) exploit parallel execution of (partial) loop iterations, which is not currently done automatically by *ParA*. Dealing with loop-independent and loop-centric parallelization in a unified manner is subject to our ongoing research.

Since the goal of the *Sambamba* project is to avoid the manual task of identifying parallel chunks and annotating them accordingly, we created the *serial elision* of each benchmark by stripping all cilk annotations. At this point, we had to drop 5 of the programs, since they are using so called cilk inlets, including early aborts, for which the serial elision cannot be created easily. The serial elision is then processed by *Sambamba* as described in Section 3.

In this evaluation, we want to assess several hypotheses. First, as we use the cilk example suite, we know that parallelism exists in the subject programs. Thus, we will compare *Sambamba*'s runtime to sequential execution, and to cilk's parallel execution.

HYPOTHESIS 1. Sambamba *is able to detect and exploit substantial parallelism from parallelizable applications.*

Second, we argued in Section 4, that brute force parallelization is often not the best option. Therefore, we implemented different dynamic dispatch schemes to dynamically choose between parallel or sequential execution of a specific function.

HYPOTHESIS 2. *Dynamic adaptation can improve the performance over brute force parallelization.*

In order to confirm our hypotheses, we report speedups of the cilk programs for different adaptation schemes in Figure 3. In addition to the 13 cilk programs, we also report the runtime of the linked list hashing program from Listing 1. All runtimes are normalized to the sequential execution of the serial elision compiled with sambamba. This sequential execution time includes starting up the virtual machine, reading the bitcode of the application, and generating machine code using just-in-time compilation. We certified that this runtime matches that of the standard lli tool. All programs were compiled and executed on a quad-core machine with an Intel i7 processor at 2.67GHz.

The figure shows the speedup using our different dispatch schemes as described in Section 4. We do not report runtime for the load dispatcher, since no other processes were running on the evaluation machine. Thus the runtime was equivalent to brute force parallelization for all programs. For the numbers marked "adaptive", the dynamic profiler was enabled possibly triggering regeneration of parallel schedules at runtime.

The runtime we considered is the wall-clock time of the full execution, including initializing the *Sambamba* runtime system, just-in-time compiling the application code, and all setup done by

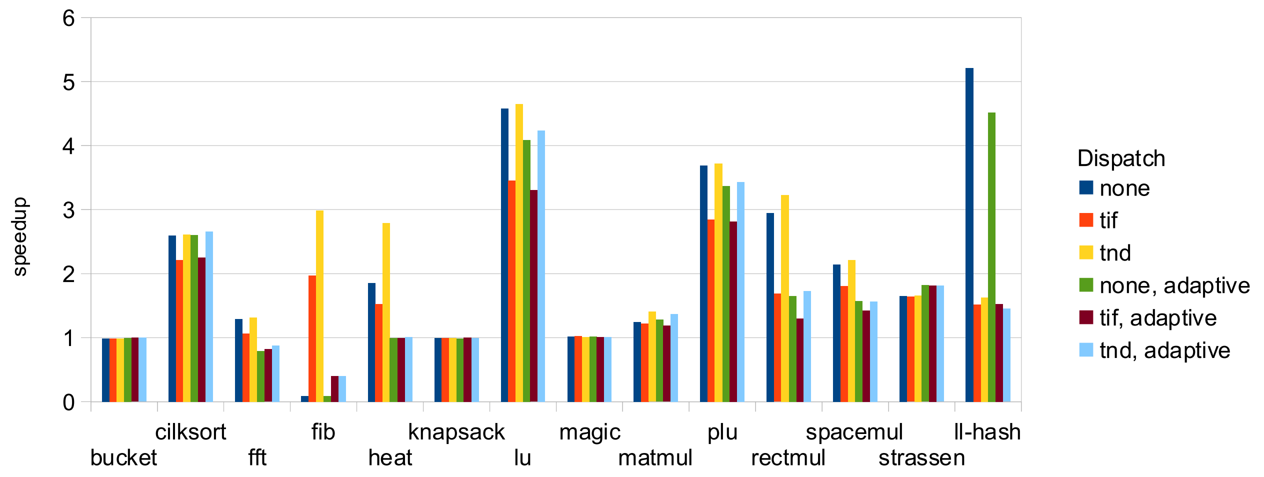

Figure 3. Runtime of programs from the cilk suite processed by *Sambamba*.

the application itself. Some of the programs do output their runtime themselves, but this would only incorporate the main processing work, e.g. sorting a big array in the case of cilksort. Of course the runtime improvement of this reported time is even bigger than the one we report, but we found it more reasonable to also include all setup work.

In the following we investigate whether our hypotheses can be confirmed to hold.

5.1 Hypothesis 1

It is clearly visible that for the majority of the programs, *Sambamba* is able to generate significant speedups. For 11 out of the 14 programs, the runtime of the parallelized program decreased significantly in comparison to sequential execution. For two programs, we even got above 4-fold speedup on the 4-core machine.

In addition to the runtimes reported in Figure 3, we also compared against a version processed and compiled by Cilk. As Cilk requires an old version of GCC (2.95), which was not readily compilable on a modern 64 bit machine, we conducted these experiments in a 32 bit environment on an eight core server. The evaluation on that machine showed that for 5 out of the 14 programs, *Sambamba* got better runtimes than Cilk. The geometric mean of the speedup achieved by sambamba over the 14 programs was still below that of cilk (1.83x compared to 2.46x). Speedups have been measured as compared to optimized sequential binaries generated with the clang compiler.

To summarize, however, *Sambamba* is able to extract a great amount of the existing parallelism.

5.2 Hypothesis 2

In four test cases (fib, heat, matmul and rectmul), adaptive dispatch improved runtime significantly over brute force parallelization. Only in one case (ll-hash), brute force was indeed the best option for the chosen input. This is because in ll-hash, the parallel tasks are not in balance: one tasks computes the hash for one element of the list, the other task recursively processes the whole rest of the list. In this case, obviously *tnd* dispatch is not appropriate because it stops parallel execution after several recursion steps. *Tif* dispatch will also execute some iterations in parallel, but once enough tasks are in flight, it stops spawning new threads, and this decision is kept for all recursive calls. So for the ll-hash case, both dispatch mechanisms will not provide scalable parallelism.

Apart from this exception, adaptation never degrades the overall performance, but in several cases it reduces the overall runtime or eliminates large overheads of brute-force parallelization.

6. Conclusion and Ongoing Work

Our *Sambamba* framework is still in an early stage. We are continuously working to make it a general platform enabling research on adaptive program optimization, in particular parallelization. The efforts described in this paper have been a first step into the direction of runtime adaptive parallelization. In particular our current implementation addresses all of the three goals mentioned in Section 1:

1. *ParA* is able to make the program execution **adapt to the environment** by taking into account profiled execution times, the number of tasks in flight or the system load.
2. *Sambamba* enables **continuous adaptation** of the running application by reacting to changing prerequisites like execution times, retriggering a new round of adaptation.
3. And *Sambamba* allows to **move parallelization related decisions to the runtime**. *ParA* uses gathered runtime information to dynamically schedule the code version deemed most appropriate for the situation at hand.

However the existing solutions leave room for improvement; in our ongoing work, we use this running system as a platform to develop and assess in-depth solutions to our goals. In the following we mention two of our most recent projects that will extend the range of *Sambamba* and the *ParA* module to incorporate more applications.

6.1 Efficient Speculation Support

Our own early studies [5] have shown that a tremendous amount of parallelization potential is hidden in general purpose applications. So far, no approach of automated parallelization has been able to sufficiently grasp that potential. Based on our studies and supported by other researchers we think that speculation support is necessary to successfully parallelize existing sequential applications.

Sambamba already includes such a speculation system in the form of software transactional memory. As the use of such a system incurs a non negligible overhead, it is only profitable in very limited cases. Also, to guide explorative adaptive parallelization, we need the speculation system to not only tell if and how often misspeculation occurs, but also where and why.

Developing a runtime efficient speculation system that is able to give precise feedback to our adaptation mechanisms is our next goal.

6.2 Generalized Automatic Adaptation

The adaptation approaches described in this paper, in particular the dispatch mechanisms in Section 4, only implicitly take into account the dispatched function. Values of arguments, or used global variables, are not considered.

In their work on a dynamic Java optimization framework, Suganuma et al. [16] have shown that continuous adaptation at runtime explicitly taking into account values of parameters and global variables can lead to performance improvements as compared to non-adaptive compilation. However they only apply adaptation based on replacing variables in the code by constants and propagating these. Method versions generated this way are dispatched by choosing a version corresponding to current parameter and global variable values. We think that the positive effect of adaptation is even higher in the context of such a heavy optimization as parallelization.

We are implementing a generalized adaptation and dispatch mechanism that employs machine learning to learn which version to choose, depending on values of parameters and global variables and the call site. This approach will allow for multiple versions of functions to exist and being chosen from automatically. The versions can be independently and continuously optimized, taking into account the dispatch criteria for the corresponding version.

7. Acknowledgments

The work presented in this paper was performed in the context of the Software-Cluster project EMERGENT[2]. 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] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[2] K.-F. Faxen, K. Popov, L. Albertsson, and S. Janson. Embla - data dependence profiling for parallel programming. In *International Conference on Complex, Intelligent and Software Intensive Systems*, CISIS '08, pages 780–785, 2008.

[3] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 21:1793–1807, 2010.

[4] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[5] C. Hammacher, K. Streit, S. Hack, and A. Zeller. Profiling java programs for parallelism. In *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, IWMSE '09, pages 49–55. IEEE Computer Society, 2009.

[6] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[7] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis and transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, 2004.

[8] C. Lattner, A. Lenharth, and V. Adve. 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 '07, pages 278–289, New York, NY, USA, 2007. ACM.

[9] J. Mak and A. Mycroft. Critical-path-guided interactive parallelisation. In *Proceedings of the 2011 40th International Conference on Parallel Processing Workshops*, ICPPW '11, pages 427–436. IEEE Computer Society, 2011.

[10] A. Raman, A. Zaks, J. W. Lee, and D. I. August. Parcae: a system for flexible parallel execution. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 133–144, New York, NY, USA, 2012. ACM.

[11] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 114–123, New York, NY, USA, 2008. ACM.

[12] V. Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. *IBM J. Res. Dev.*, 35(5-6):779–804, Sept. 1991.

[13] B. Steensgaard. Sequentializing program dependence graphs for irreducible programs. Technical report, Microsoft Research, Oct. 1993.

[14] K. Streit, C. Hammacher, A. Zeller, and S. Hack. Sambamba: Framework for adaptive program optimization. `http://www.sambamba.org`.

[15] K. Streit, C. Hammacher, A. Zeller, and S. Hack. Sambamba: A runtime system for online adaptive parallelization. In M. O'Boyle, editor, *Compiler Construction*, volume 7210 of *Lecture Notes in Computer Science*, pages 240–243. Springer Berlin Heidelberg, 2012.

[16] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a java just-in-time compiler. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 180–195, New York, NY, USA, 2001. ACM.

[17] M. Wolfe. Doany: Not just another parallel loop. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 757 of *Lecture Notes in Computer Science*, pages 421–433. Springer Berlin Heidelberg, 1993.

[2] `www.software-cluster.org`