Generalized Task Parallelism

KEVIN STREIT, JOHANNES DOERFERT, CLEMENS HAMMACHER,
ANDREAS ZELLER, and SEBASTIAN HACK, Saarland University, Germany

Existing approaches to automatic parallelization produce good results in specific domains. Yet, it is unclear how to integrate their individual strengths to match the demands and opportunities of complex software. This lack of integration has both practical reasons, as integrating those largely differing approaches into one compiler would impose an engineering hell, as well as theoretical reasons, as no joint cost model exists that would drive the choice between parallelization methods.

By reducing the problem of generating parallel code from a program dependence graph to integer linear programming, *generalized task parallelization* integrates central aspects of existing parallelization approaches into a single unified framework. Implemented on top of LLVM, the framework seamlessly integrates enabling technologies such as speculation, privatization, and the realization of reductions.

Evaluating our implementation on various C programs from different domains, we demonstrate the effectiveness and generality of generalized task parallelization. On a quad-core machine with hyperthreading we achieve speedups of up to 4.6×.

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

General Terms: Performance

Additional Key Words and Phrases: Automatic parallelization, integer linear programming, just-in-time compilation

ACM Reference Format:
Kevin Streit, Johannes Doerfert, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. 2015. Generalized task parallelism. ACM Trans. Architec. Code Optim. 12, 1, Article 8 (April 2015), 25 pages.
DOI: http://dx.doi.org/10.1145/2723164

1. INTRODUCTION

To exploit the power of modern multicore architectures on legacy software, one has to make it run in parallel. As manually parallelizing software is expensive and error-prone, automatic parallelization has always been a central research goal.

The past decades have produced several concepts and parallelization approaches. Each one is tailored to exploit parallelism found in *specific program patterns:* Given a suitable reduction analysis, any *DOALL-style loop parallelizer* can enable parallel execution of each of the three loops in Figure 1(a). Loops that carry dependencies do not qualify for *DOALL*-style parallelization. They can be dealt with by employing *DOACROSS*-style loop parallelization or *Decoupled Software Pipelining*, for instance. The manually tail-call optimized version of *quicksort* in Figure 1(c), is an example for

New Paper, Not an Extension of a Conference Paper.
Authors' addresses: K. Streit, J. Doerfert, C. Hammacher, A. Zeller, and S. Hack, Campus E1 1, 66123 Saarbruecken, Germany; emails: {streit, doerfert, hammacher, zeller, hack}@cs.uni-saarland.de.

DOI: http://dx.doi.org/10.1145/2723164

```
void kernel_bicg(double A[NX][NY],
                 double s[NY], double q[NX],
                 double p[NY], double r[NX]) {
  for (int i = 0; i < NY; i++)
    s[i] = 0;
  for (int i = 0; i < NX; i++) {
    q[i] = 0;
    for (int j = 0; j < NY; j++) {
      s[j] = s[j] + r[i] * A[i][j];
      q[i] = q[i] + A[i][j] * p[j];
} } }
```

(a)

```
void fft_twiddle(int i, int i1, COMPLEX *in,
                 COMPLEX *out, COMPLEX *W,
                 int nW, int nWdn, int r, int m) {
  if (i == i1 - 1) {
    fft_twiddle_gen1(in + i,out + i,W,r,m,nW,
                     nWdn * i,nWdn * m);
  } else {
    int i2 = (i + i1) / 2;
    fft_twiddle(i,i2,in,out,W,nW,nWdn,r,m);
    fft_twiddle(i2,i1,in,out,W,nW,nWdn,r,m);
} }
```

(b)

```
void seqquick(ELM * low, ELM * high) {
  ELM *p;
  while (high - low >= 1) {
    p = seqpart(low, high);
    seqquick(low, p);
    low = p + 1;
} }
```

(c)

Fig. 1. Different parallelizable functions: *BiCG* (a) kernel implementation (taken from the Polybench 3.2 suite); and *fft* (b) and *Quicksort* (c) implementations (adapted from the *cilksort* and *fft* programs of the *Cilk* example application suite).

such a loop. Recursive tasks, as shown in Figure 1(b), can be parallelized by classical fork-join-style *task parallelism*. Whether all these approaches can do so in a profitable way is a different question.

All these approaches work well for the form of parallelism they have been developed for; however, they are also *restricted* to these patterns. We are not aware of a single approach that would parallelize all of these three code examples, not even to speak of handling code, which embodies complex combinations of loops and recursion across several functions. Indeed, an integration of these parallelization approaches poses tremendous practical challenges, as the underlying models and assumptions are vastly different. But even if we could build a compiler that provides them all, we still would lack a joint cost model that is powerful enough to drive the choice between different kinds of parallelism.

An integrated approach, however, would be worthwhile to combine the strengths of the present approaches for parallelization. The need for integration becomes even more obvious when *speculation*, *privatization*, and *reduction* are taken into account—three techniques that have been identified frequently as being among the most important techniques for enabling parallelism. Enabling parallelism naturally increases the range of possibilities for a parallelizer to choose from. However, it is not automatically implied that such parallelism can be exploited in a profitable way. Each of those techniques introduces—potentially significant—overhead when applied. This overhead needs to be compensated for by an equally high reduction of execution time.

As a detailed example, reconsider *BiCG* in Figure 1(a). Here, depending on the chosen form of parallel execution, we have different opportunities to realize a reduction. Ignoring the possibility of performing complex iteration-space transformations, an automatic parallelizer is left with the decision of which loop to parallelize. When parallelizing the outer loop of the loop nest, the reduction induced dependence via the array *s* needs to be broken and handled specially in the parallel code. This can be done by privatizing the whole array *s*, or, for instance, by using atomic operations (or unordered atomic

sections) instead. An alternative would be to parallelize the innermost loop only. In that case only one array cell needs to be privatized in order to fix the broken reduction dependence via the array q. Making a qualified decision on the way to parallelize the code and fix the broken reduction dependencies, or even to speculate on statically unanalyzable dependencies requires one to take multiple factors into account, some of which are not known at compile time: user input, execution platform, available number of cores. All this calls for a *deep integration of parallelization approaches on a sound theoretical base, implemented on a stable and powerful platform for compile-time and runtime analysis*.

In this article, we introduce the concept and implementation of *generalized task parallelism*—a single unified framework for automated parallelization. After presenting a brief overview of our parallelization system in Section 2, our contributions, as detailed in the remainder of this article, are as follows:

(1) We present a uniform program representation based on the Program Dependence Graph (PDG) (Section 3). Relevant properties like profiling information, reduction, privatizability, and speculation opportunities are broken down to the dependence level and correspondingly represented in the PDG.
(2) Based on this representation, we *reduce the problem of parallelization to linear optimization to find local parallelization candidates* (Section 4). The formulation is independent of any special form of parallelism, and integrates central aspects of existing approaches without reimplementing them. Optimization is driven by a *cost model* derived from static profile estimates and runtime profiling information. Implementations of this approach do not rely on special code features. Loop structures, for instance, are completely transparent: Existing loops can be fully (i.e., DOALL-style) or partially (in case of carried dependencies) parallelized, while still allowing for loop-independent task parallelism.
(3) We show how to effectively generate *specialized parallel code* from the found generalized parallelism. Together with an adaptive runtime system, that can continuously reassess parallelization decisions, our parallelizer *Sambamba* is able to match the performance of specialized parallelization approaches (Section 5).
(4) We *evaluate* the presented approach (Section 6) on a set of programs from various benchmark suites, showing that generalized task parallelism
    (a) subsumes and integrates different and independent forms of parallelism;
    (b) discovers parallelization opportunities similar to those found by experts; and
    (c) produces efficient parallel code for a broad range of applications.

After relating to existing work (Section 7) and discussing technical limitations and future work in Section 8, we conclude in Section 9.

2. SYSTEM OVERVIEW

The goal of our parallelizer, *Sambamba*, is to find for each function of an application a set of *parallelization opportunities* from which at runtime the combination is chosen that best fits the execution environment. Parallelization opportunities are found in the form of arbitrary, possibly nested, regions of code amenable for parallel execution.

Sambamba consists of two parts: a *compile-time component*, performing most of the time-consuming program analyses, transformations, and scheduling offline; and a *runtime component*, building on statically gathered information and continuously collected runtime profiles to perform online adaptive optimizations.

Figure 2 gives an overview of the workflow of *Sambamba*. The application, in the form of its compiled LLVM bitcode, is read as input. This enables *Sambamba* to deal with programs written in different languages; no syntactic information is required. The resulting control flow graph of each individual function is then preprocessed and

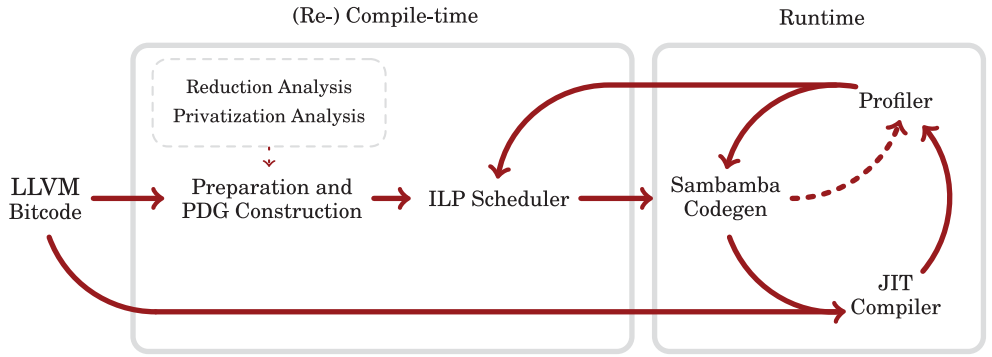


Fig. 2. Overview of the *Sambamba* parallelization system.

a PDG constructed. Also, reduction and privatization opportunities are identified and reflected correspondingly in the PDG. A scheduler based on Integer Linear Programming (ILP)[1] is used to find a set of parallelization candidates per function; it takes into account statically estimated and, if available, dynamically gathered profiling data and generates an optimal schedule with respect to the execution cost model, expressed in its constraints. The found candidates are called local paralleization candidates and reflect parallelization opportunities that are statically deemed beneficial; the decision if a local candidate will be instantiated is left to the runtime system. Note that the set of candidates may contain, but is in no way limited to, parallel loops. It may well be that the scheduler decides to execute arbitrary regions of code in parallel to each other. The cost model explicitly reflects the cost of exploiting reduction or privatization candidates. It is up to the scheduler to decide if and where such opportunities are worthwhile to realize with respect to its optimization function.

At runtime, the statically found parallelization candidates are evaluated, considering the actual execution environment; one parallel version of each function is generated for the best combination of its local parallelization candidates. To decide on the quality of a combination, a modified version of the scheduler cost function is used. Using a just-in-time compiler, this parallel version is compiled and patched into the running application. A dynamic dispatch mechanism is installed to decide, upon calls to the function, whether execution should proceed with the parallel or the sequential version. The application is continuously monitored by an efficient, sampling based profiler. The empirically gathered execution time of individual call sites allow *Sambamba* to react to changing runtime conditions.

Sambamba works in a fully automatic way and is used like a regular C/C++ compiler. Additionally, speculation hints can be given by the programmer to guide parallelization. Nevertheless, *Sambamba* in no way relies on the existence of such hints, nor on their correctness.

Note that the focus of the work described in this article is on the static component of *Sambamba*. Nevertheless, knowledge of the capabilities of the runtime system are important to understand some of the design decisions in that part. Therefore, Section 5 will give a very short overview of the dynamic capabilities of *Sambamba*, while Sections 3 and 4 will explain the static parts of *Sambamba* in detail.

[1]We use the IBM Cplex ILP solver in our current implementation. This choice is, nevertheless, not important for our approach and the solver can easily be replaced by another implementation.

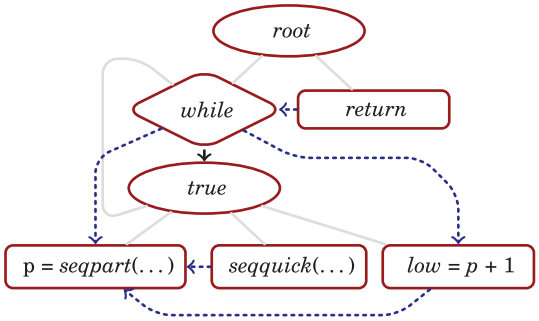


Fig. 3. The simplified PDG of *seqquick*: data dependencies are depicted by dashed arrows, control dependencies by solid arrows. Light solid lines depict parent relationship. Note how loops are represented in the PDG.

3. PROGRAM REPRESENTATION

Sambamba works solely on PDGs [Ferrante et al. 1987]. At compile time a PDG is constructed for each function. Such a PDG is kept during compilation and, if parallelism has been found, also during application runtime. The following sections explain in detail what the *Sambamba* PDG looks like and what form of extended information is stored in it.

3.1. Program Dependence Graph (PDG)

As stated by Sarkar [1991], the PDG is a perfect representation to express and analyze parallelism: It abstracts from overly restrictive implementation-dictated execution order and unveils all available parallelism by ordering instructions solely based on actual dependencies. The difficulty is to find the *right granularity of parallel execution*, as the parallelism reflected by the PDG is too fine grained, in general. Computation nodes need to be grouped to form coarser parallel tasks, costly enough to outweigh the overhead of packing and spawning.

In *Sambamba*, individual PDG nodes represent basic blocks of instructions. Before computing the PDG, basic blocks are split to isolate instructions of interest and increase the freedom to schedule them independently. Such instructions are, for example, accesses to reduction and induction variables as well as function calls.[2]

Sambamba computes data dependencies by an interprocedural, context-sensitive points-to analysis based on the Data Structure Analysis (DSA) [Lattner et al. 2007] and an array access analysis (which is beyond the scope of this article) to disambiguate array accesses in loops. Furthermore, *Sambamba* allows for, but does not rely on, user annotations to give hints to the dependence analysis. Currently, such hints are useful in the presence of recursive functions, for which DSA severely overapproximates by unifying all possible effects of a strongly connected component in the call graph.

As an example, the PDG of the *seqquick* function is shown in Figure 3. The PDG contains nodes (N) partitioned into sets of three different types:

—Regular nodes (R), depicted as boxes, represent simple instructions or basic blocks.
—Decision nodes (D), depicted as diamonds, represent basic blocks with more than one successor in the control flow graph. In the LLVM context these are basic blocks terminated by conditional branches, switches, or possibly exception throwing calls (*invokes*).

[2]Note that the restriction to the basic block level is not a limitation of the approach but instead a mere technical one.

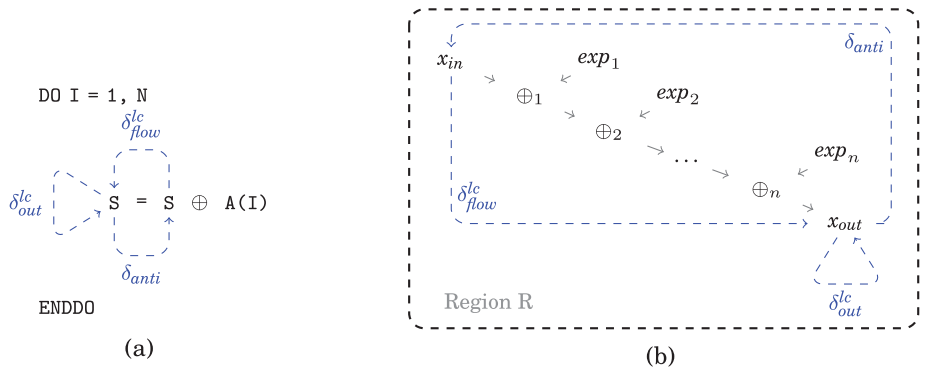


Fig. 4. Two different forms of reduction: (a) Syntactic form according to Allen and Kennedy; and (b) a general, flow-based form.

—Finally, group nodes (G), depicted as ovals, group possibly multiple nodes (called its *children*) sharing the same control condition. Each group node—except for the unique root node—is directly control dependent on exactly one decision node in the PDG.

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 dependencies between the subgraphs reachable from the group node's children 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 synchronous task parallelization.

The purpose of the scheduler as described in Section 4 is to find for each group node in the PDG a schedule of its children, representing the whole subgraph reachable from that node. Note that in this way it is possible, even natural, to generate nested parallelism. It is up to the runtime component of *Sambamba* to decide at which group nodes, and consequently also at which nesting levels, to make use of the parallelization opportunities provided by the static scheduler.

3.2. Generalized Reduction

A reduction in our sense is mostly understood as the accumulation of a value using an associative and commutative reduction operator. Together with privatization (see Section 3.3) *reduction realization* has been identified as one of the most important parallelization enabling techniques [Rauchwerger and Padua 1995; Johnson et al. 2012]. However, usually a reduction is more or less recognized syntactically, as illustrated in Figure 4(a). A loop consisting of such a successfully recognized reduction can then be transformed into a *DOALL*-style parallel loop. Usual validity criteria require the form $x = x \oplus exp$ with x not being used elsewhere in the same loop; and x not appearing in the term exp. $\oplus$ is almost always required to be an associative and commutative operator [Rauchwerger and Padua 1995; Kennedy and Allen 2002; Midkiff 2012].

In *Sambamba* we neither rely on any syntactical properties of the program nor do we restrict ourselves to specific kinds of parallelism. Instead we generalize reductions based on the following general form of a so-called *reduction chain*, as shown in Figure 4(b).

A value x is a reduction value in region R if it enters the region via an input node of x (x_{in} in the example). Examples for input nodes are load instructions or loop-carried ϕ nodes, in case R represents a loop. The value is combined with arbitrarily many expressions ($exp_1 \cdots exp_n$) using the same number of operators ($\oplus_1 \cdots \oplus_n$). It leaves the dynamic instance of R through a corresponding output node (x_{out}); for example, a store, a loop carried ϕ node, or any node used outside of R. A chain can split up and end in multiple output nodes; this, for example, happens naturally in loops with *continue* statements between different accesses to the reduction value. Furthermore, a chain can branch and join in ϕ nodes, provided x flows into each of the operators exactly once. No intermediate value of a chain is allowed to be used outside of R or outside of a valid *reduction chain* within R. Multiple independent chains on the same reduction value can exist in the same region.

A *reduction chain* is called *varying* in its containing region R if the reduction location (i.e., the x in the syntactic form $x = x \oplus exp$) can change during execution of one dynamic instance of R. The range in which the reduction location potentially varies is called the chain's *variance*.

Consider the *BiCG* implementation in Figure 1(a). The reduction in the $s[j] = s[j] + \ldots$ statement is represented by a varying chain in its containing i-loop as this statement accesses different locations ($s[0] \ldots s[NY - 1]$) in every iteration of the i loop. The $q[i] = q[i] + \ldots$ statement in contrast is represented by a nonvarying chain in the j loop.

Once all *reduction chains* for a reduction location x have been identified for a region R, all dependencies between different dynamic instances of these *reduction chains* can be relaxed, provided corresponding fix-up code is added before and after the region R.

A special case of such relaxation is the removal of the loop-carried flow dependence if R corresponds to a loop. The accesses to x in R that induced the corresponding dependence are marked in the PDG and the scheduler is allowed to break those dependencies. If it does so, we call the involved *reduction chains realized* and introduce fix-up code for those chains during final code generation, that is, at runtime.

In this work we employ the following, universally applicable scheme. For every realized reduction chain we modify the code as follows: To fix nonvarying realized *reduction chains* as well as varying ones whose variance can be evaluated before entering the corresponding region, *Sambamba* uses privatization. For varying chains with an unknown variance, it uses atomic operations without the need for privatization but at the cost of the atomic operations.

3.3. Privatization

Another very important parallelization enabling technique is privatization [Tu and Padua 1994; Vandierendonck et al. 2010]: Within a PDG subgraph R rooted in node n, a memory location x is *privatizable* iff the following conditions are met:

—on every control-flow path entering R and ending in a possible read access $l \in R$, there is at least one definitive write access $s \in R$ to x, and
—on every control-flow path from any possible write access $s \in R$ to any possible read $l' \notin R$, there is another definitive write to x on that path.

Our definition of privatizability is neither surprising nor new. But, as with reductions, we do not resolve dependencies induced by privatizable accesses prior to scheduling. Instead we annotate PDG nodes n that represent a reachable subgraph R with memory locations x to which all accesses in R are privatizable. Dependencies that enter or leave a dynamic instance of R and are induced by accesses to x are then allowed to be broken by the scheduler.

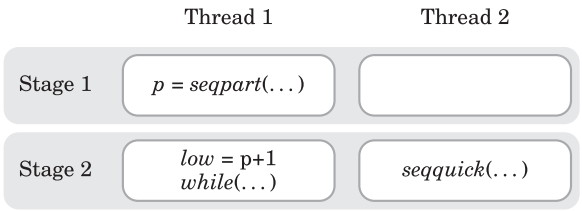


Fig. 5. Possible schedule for the group node labeled *true* of the PDG in Figure 3. This schedule represents partial parallel execution of the loop in *seqquick* of Figure 1(a).

3.4. Speculation

Speculation is an important technique that enables automatic parallelization. Unfortunately, existing techniques come at a high price in terms of runtime overhead, which needs to be reflected in the decisions of an automatic parallelizer. The problem with speculation is that its overhead inherently depends on the misspeculation rate, which in turn depends on runtime features: the number of threads/tasks running in parallel as well as the structure of the input.

Up to now, *Sambamba* does not provide a reasonably profitable speculation system. Therefore the best we can currently do is to penalize the violation of a reduction edge proportionally to the probability that it will manifest at runtime: A dependence edge that the scheduler is allowed to speculatively ignore has a source and a target potentially accessing the same memory location. The speculation is successful if at runtime the source or target statement is not executed at all. The static scheduler accounts for speculation overhead for all dependencies marked as speculatively ignorable, that the scheduler decides to break. It amounts to a value that grows linearly in the size of the commonly accessed values multiplied by the relative execution frequency of the source and target statements, respectively.

In our evaluation in Section 6 we do not rely on speculation.

4. STATIC SCHEDULING

We formulate an integer linear program to compute a *local* schedule for each individual group node in the PDG. The intuition behind the following ILP formulation is to map the children $I_g = \{g[i] \mid i \in \{1 \dots n_g\}\}$ of a group node g onto a two-dimensional grid. One dimension in this grid corresponds to time and is subdivided into *stages* (S_g) of execution; the other axis corresponds to placement and is subdivided into *threads* (T_g). $|T_g|$, the maximum number of parallel threads that the generated schedule will use, is a constant parameter to the ILP scheduler. Nodes will possibly execute in parallel at runtime iff they are placed in different threads of the same stage. The optimization goal of each ILP is to minimize the latency for its corresponding PDG group node.

Figure 5 shows a possible result of solving the ILP corresponding to the PDG group node labeled *true* in Figure 3. Stage 2 is a parallel stage in this schedule, spawning off the recursive call to *seqquick* while proceeding with the next iteration of the loop, which is represented by its PDG node labeled with the while statement in thread 1 of stage 2. A schedule representing a parallel loop is called a reentrant schedule. This example also shows how the scheduler transparently parallelizes loops without any special treatment. A schedule for a *DOALL* loop, for instance, would in its simplest form contain one stage with two threads of which one contains the loop body and the other the computation of the induction variable (if present).

As described in Section 3, dependencies between subgraphs can potentially be relaxed by making use of privatization, reduction, or speculation opportunities. Such relaxation allows the scheduler to execute the source and target of a dependence in parallel to each

other, provided, corresponding fix-up code is added. The potential overhead introduced by such countermeasures is encoded in the ILP cost function.

4.1. Prerequisites

In favor of a shorter notation, we assume $g \in G$; $r \in R$; $d \in D$; $i, j \in \{1 \dots |I_g|\}$ in the following discussion. Before constructing an ILP for each group g of a PDG, we compute estimates of accumulated execution costs for each PDG node as follows:

—The size $\|r\|$ of a regular node r is computed by traversing the contained instructions (remember that a regular node in our case represents a basic block), and accumulating their individual cost. For this purpose we statically estimate the cost of arithmetic instructions; the cost of memory instructions is taken into account by assessing the size of written, read, or copied data if statically possible. Costs of call instructions (call sites) are taken from dynamic call site profiles of earlier runs of the application, if such profiles are available, or estimated: If the called function is statically known, the call graph is traversed and the cost of transitively called functions accumulated. If the call is indirect, that is, the called function is not statically known, we assume high cost for the call. This is in favor of parallelization and leaves it to the runtime code generation to use call site execution time profiling to find out if the assumption was beneficial or not.

—$\|d\|$, the execution cost of a decision node d, is defined as $\sum_i (\|d[i]\| * freq(d_i)) + |d|$, where $freq(d_i)$ is the frequency of d selecting child node $d[i]$ for execution, and $|d|$ is the nonaccumulated size of d (which is computed in a similar way to the cost of regular nodes). Frequencies are statically estimated or read from profiling information persisted during earlier runs of the application. *Sambamba*'s runtime system is able to collect such branch profiles (see Section 5.3).

—$\|g\|$, the size of a group node g, is defined as $\sum_i \|g[i]\|_g$.

4.2. ILP Formulation

For each group node g of the PDG, a directed acyclic graph $DAG_g := (I_g, \Delta_g)$ describes the data dependencies and possible conflicts between the children of g. Its nodes are the children of g (i.e., I_g); Δ_g is the set of edges.

An edge $(i \to j)_g \in \Delta_g$ has the meaning of its source $g[i]$ depending on or conflicting with its target $g[j]$ in g. It has associated communication cost $\|(i \to j)_g\|$, being an estimate of data to be communicated if $g[i]$ and $g[j]$ execute in different stages. An edge is called fulfilled by a schedule if according to the schedule its target is executed before its source.

$\Upsilon_g \subseteq \Delta_g$ is the set of edges requiring speculation support in the source and target thread of the dependence if the scheduler breaks it; $\Omega_g \subseteq \Delta_g$ and $\Psi_g \subseteq \Delta_g$ are the reduction and privatization ignorable edges, respectively. Note that Υ_g, Ω_g, and Ψ_g are not necessarily disjoint. One edge might require multiple fix-up mechanisms in order to be breakable.

To reduce the number of constraints, we precompute the set of transitive edges $\Theta_g \subseteq \Delta_g$, thus $(I_g, \Delta_g \setminus \Theta_g)$ is the transitive reduction of DAG_g. We call $\widetilde{\Delta_g} := \Delta_g \setminus (\Upsilon_g \cup \Omega_g \cup \Psi_g)$ the set of *unbreakable dependencies*.

For the sake of brevity, we assume that $s \in \{1 \dots |S_g|\}$ and $t \in \{1 \dots |T_g|\}$. Table I introduces the variables used in the ILP formulation. Note that the number of variables is quadratic in I_g since $|S_g| \leq |I_g|$ and $|T_g|$ is a constant.

Figure 6 shows the final ILP formulation. The used objective function minimizes the critical path execution time ($\sum_s \gamma_g[s]$) while penalizing the use of multiple stages and threads, as well as interthread communication. Additionally, broken dependencies marked as resolvable by privatization, reduction, and speculation are punished. Each

Table I. Variables used in the ILP for a Group Node g

$\gamma_g[s] \in \mathbb{N}$	:=	critical path length of stage s
$\varphi_g[s] \in \mathbb{B}$	:=	1 iff stage s is nonempty
$\sigma_g[s,t] \in \mathbb{N}$	:=	size of thread t in stage s
$\varphi_g[s,t] \in \mathbb{B}$	:=	1 iff thread t in stage s is filled
$\sigma_g[i,s,t] \in \mathbb{N}$	:=	size of $g[i]$ in stage s, thread t
$\chi_g[i,s,t] \in \mathbb{B}$	:=	1 iff $g[i]$ is placed in stage s, thread t
$par_g[i,j] \in \mathbb{B}$	:=	1 iff $g[i]$ and $g[j]$ execute in parallel

Minimize

$$\sum_s \left(\gamma_g[s] + \varphi_g[s] * SInitOvhd + \sum_t \left(\varphi_g[s,t] * TInitOvhd \right) \right) + ComCost_g + RelaxP_g$$

where

$$ComCost_g := \sum_{(i \to j)_g \in \Delta_g} \sum_s \sum_t \left(abs\left(\chi_g[i,s,t] - \chi_g[j,s,t] \right) * \frac{\|(i \to j)_g\|}{2} \right)$$

$$RelaxP_g := \sum_{(i \to j)_g \in \Upsilon_g \cup \Omega_g \cup \Psi_g} \left(par_g[i,j] * \left(SpecP_g[i,j] + RedP_g[i,j] + PrivP_g[i,j] \right) \right)$$

SInitOvhd and *TInitOvhd* are ILP constant estimates of stage and thread initialization overhead respectively.

subject to the following constraints

Constr. 1 (Unique Placement)

$$\forall i\colon \sum_s \sum_t \chi_g[i,s,t] = 1$$

Constr. 2 (Dependence Order 1)

$$\forall (i \to j)_g \in \widetilde{\Delta_g} \setminus \Theta_g\colon SD_{i,j} * |T_g| - TD_{i,j} \geq 0$$

Constr. 3 (Dependence Order 2)

$$\forall (i \to j)_g \in \widetilde{\Delta_g} \setminus \Theta_g\colon SD_{i,j} * |T_g| + TD_{i,j} \geq 0$$

Constr. 4 (Size Placement Connection)

$$\forall i \forall s \forall t\colon \sigma_g[i,s,t] = \chi_g[i,s,t] * \|g[i]\|$$

Constr. 5 (Parallel)

$$\forall (i \to j)_g \in \Delta_g\colon par_g[i,j] * |T_g| \geq abs(TD_{i,j}) - |T_g| * abs(SD_{i,j})$$

Constr. 6 (Thread Filling)

$$\forall s \forall t\colon \sum_i \chi_g[i,s,t] \leq \varphi_g[s,t] * |I_g|$$

Constr. 7 (Stage Filling)

$$\forall s\colon \sum_t \varphi_g[s,t] \leq \varphi_g[s] * |T_g|$$

Constr. 8 (Thread Size)

$$\forall s \forall t\colon \sigma_g[s,t] = \sum_i \sigma_g[i,s,t]$$

Constr. 9 (Critical Path)

$$\forall s \forall t\colon \gamma_g[s] \geq \sigma_g[s,t]$$

Constr. 10 (Speculation Order)

$$\forall (i \to j)_g \in \Upsilon_g\colon STD_{i,j} \geq 0$$

where

$$SD_{i,j} := \sum_s \sum_t \left(s * \left(\chi_g[i,s,t] - \chi_g[j,s,t] \right) \right) \quad \textbf{and} \quad TD_{i,j} := \sum_s \sum_t \left(t * \left(\chi_g[i,s,t] - \chi_g[j,s,t] \right) \right)$$

$$\textbf{and} \quad STD_{i,j} := \sum_s \sum_t \left(\left(s * |T_g| + t \right) * \left(\chi_g[i,s,t] - \chi_g[j,s,t] \right) \right)$$

Fig. 6. Objective function and constraints used in the ILP formulation for group node g.

used stage is modeled to introduce overhead (*SInitOvhd*), which is motivated by the setup cost of synchronization mechanisms at runtime. Each used thread causes runtime overhead (*TInitOvhd*) due to the cost of setting up and spawning parallel tasks.

ComCost accounts for introduced interthread and interstage communication by adding the statically estimated communication cost $\|(i \to j)_g\|$ per boundary crossing dependence. This estimate is solely based on the communication volume (i.e., the static size of the communicated data) in the current implementation. A dependence

$(i \rightarrow j)_g \in \Delta_g$ is considered boundary crossing if, and only if, $\chi_g[i, s, t]$ and $\chi_g[j, s, t]$ differ in the current solution of the ILP.

The speculation penalty cannot model the actual overhead of speculation statically: the main cause for overhead is frequent rollbacks and reexecutions, which are inherently input dependent. Instead, the speculation penalty is approximated by an ILP-constant overhead $SpecP_g[i, j]$ per speculatively ignored dependence crossing a thread boundary. It is computed based on the execution frequencies of the PDG nodes causing the conflict represented by the dependence. *RelaxP* accounts for this by adding $SpecP_g[i, j]$ for each speculatively ignored edge $(i \rightarrow j)_g \in \Upsilon_g$.

Reduction $RedP_g[i, j]$ and privatization $PrivP_g[i, j]$ penalties are treated similarly to the speculation penalties. But the individual penalty per broken dependence differs: In the case of privatization, the penalty grows linearly with the size of the value to privatize. Similarly, the reduction penalty grows linearly with the size of the reduction variable. In the case of a varying chain, this size is multiplied by the chain's variance.

The constraints of the ILP can be partitioned into two groups: The first group is formed by the Constraints 1, 2, 3, and 10, which are used to model legality constraints to preserve the semantics of the program:

—Constraint 1 ensures that each node is scheduled exactly once. It guarantees *unique placement* of nodes.
—Constraints 2 and 3 ensure for each unbreakable dependence $(i \rightarrow j)_g$, which is not marked transitive, that execution of $g[j]$ precedes execution of $g[i]$. This means that $g[j]$ is executed in an earlier stage than $g[i]$, or in the same stage and thread. Note that $SD_{i,j}$ denotes the stage distance between the placement of $g[i]$ and $g[j]$. The constraints ensure that $SD_{i,j}$ is non negative, and if it is zero, then also $TD_{i,j}$ (the thread distance) must be zero.
—Constraint 10 ensures that for each speculatively ignorable dependence $(i \rightarrow j)_g, g[j]$ is executed in an earlier stage than $g[i]$, or in the same stage, but possibly different thread with a lower number. The fact that there still are restrictions on speculatively ignored dependencies is due to the conflict detection and recovery mechanism of the runtime system. It allows for conflict detection only between different threads of the same stage. Thus, $g[j]$ is not allowed to be placed in a later stage than $g[i]$. The requirement that $g[j]$ needs to be in a thread with a lower number t than $g[i]$ is necessary to guarantee a noncyclic commit order between the threads.

The second group is formed by the Constraints 5–9, which model execution cost and are used in the objective function:

—Constraint 5 defines the $par_g[i, j]$ variables that are used in the cost function to penalize dependencies broken using speculation, privatization, or reduction.
—Constraints 6 and 7 define the $\varphi_g[s, t]$ and $\varphi_g[s]$ variables to reflect if a thread or stage is filled, i.e., contains at least one node.
—Constraint 8 models the size $\sigma_g[s, t]$ of a thread as the sum of sizes of contained nodes.
—Constraint 9 models the critical path length $\gamma_g[s]$ of a stage as the size of the largest contained thread: it is pulled down as it is used in the objective function, but guaranteed to stay larger than the size of any contained thread.

The remaining Constraint 4 connects both groups of constraints by relating the Boolean variables $\chi_g[i, s, t]$ to the corresponding size variables $\sigma_g[i, s, t]$. Further, it defines the size of node $g[i]$ as being the ILP constant $\|g[i]\|$.

Note that no further constraints are required for reduction and privatization resolvable edges. A dependence that is resolvable by privatization or reduction, can, in terms of legality, simply be ignored. It only needs to be reflected correspondingly in the

cost of the resulting schedule. This is taken account for in the *RelaxP* definition of the cost function. The number of constraints is quadratic in the number of child nodes per group.

Further, remember that Constraint 1 guarantees that each node is uniquely placed, that is, not duplicated among parallel threads. It is possible to relax this restriction for nodes that are safe to clone: Every node that is guaranteed not to execute any observable side effect (program termination, system calls or memory writes) can be duplicated. However, such relaxation severely increases the complexity of other constraints whose formulation relies on every node to be placed exactly once.

The ILP solver can be initialized with the sequential schedule: $\chi_g[i, s, t]$ is set to 1 for $s = 0$ and $t = 0$, 0 otherwise. Providing an initial solution effectively speeds up the optimization process in practice.

4.3. A Note on Intercore Communication

Recent successful work on parallelizing irregular applications [Raman et al. 2008; Campanoni et al. 2012] has shown that, depending on the form of parallelism, intercore communication latency is a limiting factor for successful parallelization. Helix [Campanoni et al. 2012, 2014], for instance, is able to achieve impressive speedups on irregular applications without even relying on speculation. The approach, however, strongly relies on efficient intercore communication, as loop-carried dependencies have to be communicated between every pair of succeeding loop iterations, which are executed on different cores in a round-robin fashion. By clever scheduling, the communication latency can be hidden to some degree but the scalability of the approach by design is limited by the communication latency.

The influence of intercore communication on the successful parallelization using *Sambamba* is limited in several ways. Note that intercore communication only happens when spawning a parallel task and only from the spawning to the spawned task. This communication is minimized in the ILP formulation by including the *ComCost* in the optimization function. Furthermore, task blocking (see Section 5.2) greatly reduces the amount of necessary communication by executing multiple successive instances of a task, for example, loop iterations, on the same core. Recomputable values, like, for instance, induction variables, are communicated only once per block instead of once per task instance. Values that are needed by multiple task instances, for example, loop-invariant live-in values, are communicated once per block at most. For the remaining, strictly necessary communication, a work-stealing task scheduler[3] in the spirit of cilk is employed by the *Sambamba* runtime to effectively make use of the cache hierarchy and minimize necessary intercore communication; in particular, for nested parallelism.

4.4. Scheduling Time

As solving integer linear programs is NP-hard in general, our scheduling approach often takes a considerable amount of computation time. Remember that the described *ILP* is solved for each group of equally control-dependent nodes and its complexity grows quadratically in the number of nodes (or linearly in the number of dependence edges) in such a group. This consequently means that the complexity of parallelizing an application in *Sambamba* is dominated by its maximum number of equally control-dependent nodes, which does—like the average size of a basic block—not necessarily correlate with the program size. The latter only linearly influences the complexity.

Furthermore, although an increasing number of dependence edges does indeed increase the number of constraints, this does not mean that the solving time dramatically

[3]The *Sambamba* runtime system relies on the dynamic scheduler of *Intel TBB*.

Table II. Characteristics of the Programs used to Evaluate *Sambamba*. They cover a broad range of domains and parallelization schemes. Some of them require privatization or reduction recognition and handling

Benchmark	Suite	*SLOC*	$N_{max/avg}$	$E_{max/avg}$	Enabling Techniques: Priv	Enabling Techniques: Red	Applied Scheme: Loop (full)	Applied Scheme: Loop (partial)	Applied Scheme: Task
alignment	BOTS	612	93/9.86	128/9.27	✓	✓	✓		
cilksort*	Cilk	387	22/8.58	23/7.19				✓	✓
fft	Cilk	3168	63/8.92	161/10.93	✓	✓	✓		✓
blackscholes	Parsec	393	24/7.38	30/6.87			✓		
BiCG	Polybench	1586	31/9.15	37/9.50		✓	✓		
gesummv	Polybench	1582	24/7.88	31/8.08		✓	✓		

increases. This is due to the fact that an increasing number of dependencies leaves less freedom to the scheduler. IBM Cplex, the scheduler we use, is able to dramatically reduce the size of the ILP before actually solving the corresponding LP.

In practice we found that for the majority of instances (>80%) an optimal solution is found in less than 10s. To counter much longer execution times we implemented several means to limit their influence.

As can be seen in Table II, the most complex of the benchmarks is *fft*. Its highest number of dependencies per group is 161 dependence edges after transitive reduction (324 edges before). Running the ILP until the optimum is proved, takes 4290s. However, the optimal solution is found after 15s, a solution within a 10% range of the optimal solution after 20s.

In all our benchmarks, the solution could not be significantly improved after 3min, so we assume that after a timeout of 3min the best solution found so far is close enough to the optimum and interrupt the solver. This timeout is configurable.

Additionally, the generated schedules of each function are written to disk for reuse on the same machine. This greatly reduces compilation time during frequent recompilations on a developer machine—changed dependencies (and consequently, PDGs) of the application lead to rescheduling of affected functions only. Optionally, the schedules can also be cached in a shared *schedule cloud*. During idle periods the cloud server can always take partial (i.e., feasible but not optimal) solutions and improve them toward an optimal one.

Note that the techniques described in this section are of purely practical relevance and are not critical to the approach in general.

5. RUNTIME SUPPORT

The form of parallelism found by the static parts of *Sambamba* is of general nature. To be as profitable as possible, however, the finally generated code can be specialized to exploit features of the program that support efficient parallel execution. Such features include a statically known or at least loop-invariant iteration range, or the recursion scheme of parallelization candidates, for example. *Loop blocking* and *adaptive dispatch*, which we shortly describe in this section, are two important techniques to exploit them. Note that we deliberately keep their description short as we want to focus on the static parts of *Sambamba* in this article. Nevertheless, it is important to know of their existence to understand how the results in Section 6 are achieved.

5.1. Adaptive Dispatch

In the current implementation, *Sambamba* keeps one parallel version of each function in addition to the sequential version. By monitoring the execution of the program, *Sambamba* chooses which of the functions to dispatch. The dispatch criteria are based on the current system load (*load*), the utilization of the dynamic task queues (i.e., the number of tasks in flight, *tif*) or the nesting depth of parallel tasks (*tnd*). The latter one is of particular importance when recursive functions (like in the *fft* example in Figure 1(b)) are parallelized. A naive approach might excessively spawn parallel tasks at each recursion level, even though the work usually decreases and it is not profitable from a certain depth on. When parallelizing such a code manually, this issue is usually solved by introducing a parallelization threshold—a specialization of the source code that is not necessary using an adaptive runtime system such as *Sambamba*'s.

The *Sambamba* runtime system switches to the sequential version of a function in case any of the following conditions is fulfilled:

—the system is already more than 90% utilized (*load*),
—more tasks than two times the number of cores available in the system are waiting for execution in the global task queue (*tif*), or
—the nesting level of parallel execution is higher than $log_2(\#Cores) + 1$ (*tnd*).

5.2. Dynamic Blocking

Another important feature of the *Sambamba* runtime system aims at increasing the size of the parallel task. The overhead of enqueuing parallel tasks easily outweighs the actual work to be done in the tasks. This often occurs for parallelized loops doing little work, like, for example, in the *BiCG* example in Figure 1(a). The reentrant parallel section for such a loop contains exactly one reentrant task representing the iteration part of the loop, and thus contains the loop-carried dependencies. One or several nonreentrant tasks in the same section represent parallel work to be spawned in each iteration. If these tasks only contain a small amount of code, like loading a value from an array and performing a reduction operation, then the overhead of handling the parallel tasks nullifies the benefit of parallel execution.

Dynamic blocking increases the task size by dynamically aggregating a number of parallel tasks, before enqueuing it as one batch task. This saves most of the cost of scheduling tasks doing negligible work. Reduction additionally profits from another benefit, independent of the used fix-up approach: For privatization, the private copy only needs to be determined once per batch task; for atomic operations, we only need to update the shared location once.

In contrast to other techniques like recursive range splitting, dynamic blocking is more general because it does not need to know the iteration range of the loop before it is entered. The downside is that the thread that collects the tasks before spawning them in a batch may become the bottleneck. Therefore, if the loop iteration range is loop invariant, that is, known before entering the loop, the *Sambamba* runtime system produces code that immediately distributes loop execution equally among available threads, which basically corresponds to a one-level (i.e., nonrecursive) range splitting.

5.3. Runtime Profiling

The static scheduler of *Sambamba* decides for parallel execution if in doubt and relies on the runtime system to take runtime information into account to drop unprofitable candidates. Consequently, the runtime system needs to collect relevant profiling information.

In particular, the *Sambamba* runtime system collects two kinds of profiles:

—*Call-site execution times*, which is basically a context-sensitive form of average method execution time.
—*Branch profiles*, used to estimate the execution cost of decision nodes and to derive loop trip counts.

In order to minimize the introduced runtime overhead, profiling is only enabled for relevant code parts: Only call sites contained in statically determined candidate regions of parallel execution are profiled. Branch profiles are only collected for functions containing at least one parallelization candidate.

Furthermore, not every single invocation of a target function needs to be profiled. If enough samples are already available and profiling information is stable, the likelihood to learn something fundamentally new is low and profiling is disabled with a higher probability. Edler von Koch and Franke [2014] have shown that profiles, in particular data dependence and control flow related ones, stabilize quickly.

6. EXPERIMENTAL EVALUATION

For a detailed evaluation of the effectiveness and generality of *Sambamba*, we chose six programs from different domains and different benchmark suites. Characteristics of the chosen programs are given in Table II. The table lists for each program its source benchmark suite (*Suite*), the number of source lines of code (excluding empty lines and comments, *SLOC*), the maximum and average number of equally control-dependent nodes ($N_{max/avg}$) and dependence edges ($E_{max/avg}$), as well as the necessary enabling techniques and the applicable parallelization schemes.

We classify parallelization candidates into three schemes as applied by existing parallelization approaches:

(1) *Loop (full)* basically corresponds to DOALL-style parallelism in which loops without any loop-carried dependencies (except for those induced by induction variable computation) are parallelized.
(2) *Loop (partial)* corresponds to loops with loop-carried dependencies of which at least parts can be parallelized. Different parallelization approaches exist that can deal with such loops. Examples are DOACROSS, DSWP, or Helix.
(3) *Task* represents loop-independent parallelism as can be expressed, for instance, in the Cilk or Cilk++ languages. This form of parallelism is also known as fork-join parallelism.

Again, note that *Sambamba* does not *explicitly* exploit the mentioned forms of parallelism. Also it does not implement or include a specialized approach for any specific pattern of parallelism. The parallelization scheme of *Sambamba* is solely based on dependencies and does not take any particular program structure into account. The resulting parallel code produced by *Sambamba* nevertheless may have been produced by more specialized approaches falling into the previously mentioned categories. The purpose of this classification is to show that *Sambamba*, by abstracting from the program structure, *implicitly* exploits such well-known patterns of parallelism.

The programs have been chosen as representatives for their particular style of parallelism as reflected in their originating benchmark suite. Later in this section we will additionally show the results of *Sambamba* applied to all programs from the *PolyBench* benchmark suite as well as most applications of the *Cilk* example suite.

For each of the detailed evaluation subjects we compare *Sambamba* against an approach that ships with or is usually evaluated on this benchmark. We evaluate four different configurations of *Sambamba*: with runtime dispatch enabled, with loop blocking enabled, both runtime techniques enabled, and none of them enabled. Note

that the latter configuration still makes use of runtime profiling information to choose from different parallelization possibilities and form one parallel version per function.

We are assessing the following hypotheses in this evaluation:

(1) *Sambamba* identifies and leverages different forms of parallelism;
(2) *Sambamba* effectively makes use of privatization and reduction recognition; and
(3) *Sambamba* creates efficient parallelized code over a broad range of applications.

The experiments were performed on a *Intel Core i7 920* quad-core CPU with 2.67GHz, 8MB cache, and *Hyperthreading*. The *LLVM*-based approaches (*Sambamba* and *Polly*) as well as the sequential baseline were compiled with *clang 3.4.2*; other approaches, in particular the OpenMP versions, with *gcc 4.9.1 (pre-release)*.

Note that, as mentioned earlier, *Sambamba* makes use of statically estimated as well as dynamically collected profiling information. It does so during static scheduling of parallelization candidates (see Section 4.1), as well as at runtime, when selecting between different candidates or combinations thereof. For our evaluation, we did *not* take runtime profiling information into account during static scheduling, which is solely based on static estimates. The candidate selection at runtime, however, averages the profiles collected during the current and earlier runs of the binary and takes them into account. It is therefore able to use profiling information on actual inputs.

Of course, runtime profiling itself imposes overheads that need to be minimized to keep profits high. Different schemes can be thought of to do so, including statistical profiling (see Section 5.3). As we do not evaluate the runtime system of *Sambamba*, we assume the availability of runtime profiles for free. To simulate this, we enable runtime profiling during the first benchmark run. The collected information is taken into account in all future runs.

Polybench/C 3.2 by Pouchet [2012] contains mathematical, loop-based programs, which are amenable to polyhedral loop optimizations. We compare the performance against *Polly*, the polyhedral optimizer of the *LLVM* compiler framework. Note that the form of parallelism exploited by typical polyhedral optimizers is significantly different from what *Sambamba* does: Big improvements in terms of execution time are achieved by optimizing for cache locality, a goal that *Sambamba* does not currently share. Nevertheless, we consider the benchmarks chosen from this domain important as they show highly nested loops with very small bodies, which do not justify the overhead of spawning parallel tasks for every instance.

The programs taken from the *Cilk* [Blumofe et al. 1995] example suite are compared against optimized execution in *Cilk*. For the sequential reference we compiled the so-called *serial elision*[4] of those programs using the *clang* compiler, in order to avoid any overhead induced by the *Cilk* runtime system. The *BOTS* suite of Duran et al. [2009] contains one sequential and several manually parallelized program versions containing *OpenMP* annotations. The best performing *OpenMP* variant is used as reference in all cases. Parsec [Bienia et al. 2008] contains handcrafted versions of each program, parallelized using *OpenMP*, *Intel TBB*, and *native POSIX threads*. Again, we compare against the best of those versions.

Figure 7 shows the result of the evaluation. All numbers are normalized against an optimized sequential program version compiled with *clang*.

In the evaluated programs, *Sambamba* detects all the program locations that were parallelized by the domain experts. Also, *Sambamba* decreases the runtime substantially in all cases. We checked that the reported speedups are significant with a *confidence of 99%* according to the *Speedup-Test* [Touati et al. 2013]. This shows both the *generality* and *effectiveness* of our approach.

[4]Deleting *Cilk* language constructions (spawn, sync, ...).

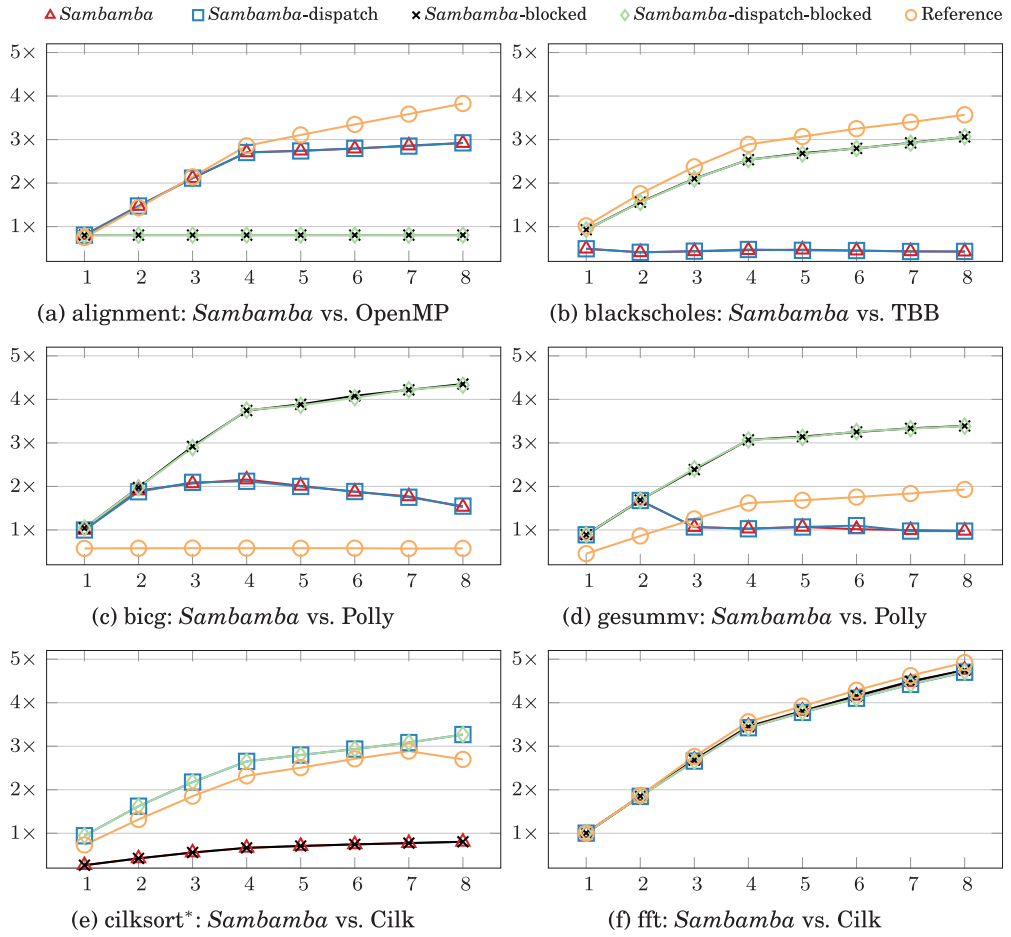


Fig. 7. Evaluation of *generalized task parallelism* (*Sambamba*) on six programs from different domains, containing different styles of parallelism. The x axis shows the number of threads, the y axis compares *speedup over sequential execution*. Parallelization in *OpenMP*, *Cilk*, and *Intel TBB* is done manually by experts; *Polly* and *Sambamba* parallelize automatically.

In the *BiCG* (7c) and *gesummv* (7d) programs from the PolyBench suite, *Sambamba* outperforms *Polly*, the specialized tool for those kinds of programs. We see that blocking significantly improves the performance of *Sambamba*. Dispatch has no benefit, but also does not introduce any overhead.

In the case of *BiCG Polly* there are mainly two problems:

—*Polly* works on basic block level and is unable to split blocks on demand. Both statements of the innermost loop (see Figure 1(a)) share the same basic block, which consequently induces loop-carried dependencies over both containing loops.
—*Polly* is unable to deal with reductions and therefore misses an important opportunity of parallelization.

The slowdown of *Polly* in this benchmark comes from the fact that it not only misses the profitable parallelism, but also parallelizes the loop initializing the s array to 0, which is not profitable.

Concerning *gesummv*, *Polly* finds the right location to parallelize but the generated code based on *OpenMP* is unable to profitably exploit the parallelism. The speedup would be higher, if *Polly* would additionally vectorize the generated code, which it does not in this particular case.

In *alignment* (7a) and *blackscholes* (7b), the efficiency of the *Sambamba*-parallelized versions fall behind the handcrafted versions using *OpenMP* or *TBB*. This is caused by two factors: First, the *OpenMP* and *TBB* programs are compiled with gcc, which creates more efficient code in these cases. Second, the *Sambamba* runtime system introduces overhead for allocating heap space for the parallel tasks, and for privatization and reduction locations. This overhead is significantly higher than that of the reference systems.

We see that *alignment* profits from neither blocking nor runtime dispatch. Indeed, blocking harms performance. This is caused by the parallelized loop not having enough iterations to reach the chosen block size. Blocking thus effectively sequentializes execution. If the choice to enable blocking is left to the *Sambamba* runtime system, it therefore disables it as it observes that the iteration count is too low for blocking to be profitable. *Sambamba* performance on *blackscholes* in contrast heavily benefits from blocking.

The *alignment* program is especially interesting, because parallelizing the main loop in the *pairalign* function requires privatizing 15 variables at different loop nesting levels. This is encoded in the *OpenMP* annotations; if one of them is missed by the developer, the executable might produce incorrect results. *Sambamba* determines them automatically and produces corresponding code without human guidance.

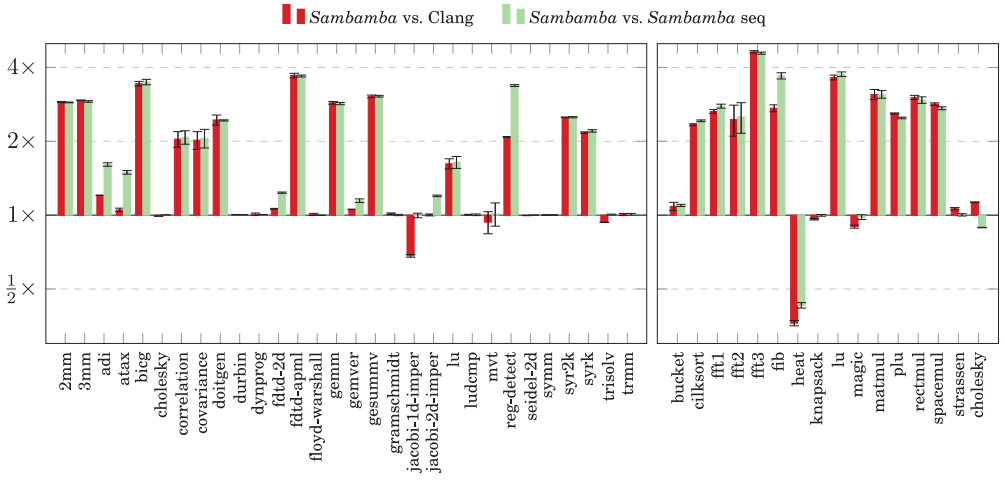
The *fft* (7f) and *cilksort** (7e) programs implement *fast Fourier transform* and a standard *mergesort*. Note that the version of cilksort contained in the Cilksuite performs a switch from *mergesort* to *quicksort* at a hard coded array size boundary. Cilk and *Sambamba* (without runtime dispatch) both profit from this boundary when automatically parallelizing as it effectively causes execution to switch from parallel to sequential once the problem size drops below a given size. As for regular targets of a parallelizer such help cannot be expected; we removed this boundary for our benchmarks (hence the * in *cilksort**). *Sambamba* makes placing such somewhat artificial boundaries superfluous.

As mentioned earlier, *Sambamba* can make use of dependence annotations given by the programmer. Like essentially all applications of the Cilk example suite, *fft* and *cilksort* mainly consist of recursive functions. As described in Section 3.1, *Sambamba* profits from user-provided annotations in such cases and we manually annotated relevant parts. The idea is similar to that of Vandierendonck et al. [2010]: hints are only used to improve dependence information while parallelization stays fully automatic.

Both *cilksort** and *fft* do not profit from blocking as the dominating parallelism does not stem from parallel loops. *fft* does not profit from runtime dispatch. This is mainly due to a highly specialized and hand-optimized implementation that switches over to specialized implementations to solve smaller subproblems: Specialized implementations that are not parallelized. This corresponds, just like in the original version of *cilksort*, to an implicit switch from parallel to sequential execution.

*cilksort**, however, greatly benefits from runtime dispatching. Indeed only with runtime dispatching is it able to achieve any speedup. In that case it constantly outperforms Cilk.

Note that a significant slowdown can be observed for *Sambamba* without dispatching, even with only one thread. This is because the overhead of creating and scheduling parallel tasks is introduced at every recursion depth and for every problem size. For the smaller problem sizes this overhead outweighs the actual productive work. Additionally, task stealing will hurt data locality, leading to the observed slowdowns for multithreaded execution.



(a) Polybench (b) Cilksuite

Fig. 8. Evaluation of *generalized task parallelism* (*Sambamba*) on the Polybench 3.2 (a) and Cilksuite (b) with eight threads on a quad core with Hyperthreading.

Sambamba with runtime dispatch is able to fully match the parallelization decisions of the manually crafted implementations; the execution time, with runtime dispatch, is comparable to execution with Cilk for both benchmarks.

Apart from the detailed benchmark evaluation shown in Figure 7 we evaluated *Sambamba* on the whole *Polybench/C 3.2* Suite [Pouchet 2012], as well as most of the applications from the suite of Cilk [Blumofe et al. 1995] example applications. The results are shown in Figures 8(a) and 8(b),[5] respectively.

The experiments have been conducted on the same machine as the previous experiments. Speedups are relative to an optimized binary produced by clang and sequential execution in the *Sambamba* framework. The latter is to demonstrate the speedup obtained by parallelization alone and ignores overhead introduced by just-in-time compiled execution, for instance. Another reason for differences in the numbers is the potential lack of interprocedural optimization of parallelized functions, which are not performed in order to maintain the freedom to exchange functions at runtime. If, for instance, a parallelized function is inlined at compile time, exchanging this function at runtime is without effect.

In the case of Polybench, the reported speedups are obtained using the timing measurement facilities of the benchmark suite, running on the large input set. In particular, the speedup refers to the main computational kernel of each benchmark and not the whole application.

As explained earlier, for the cilk applications the serial elision is computed, annotated with dependence hints, and automatically parallelized. We dropped four applications (*ck*, *game*, *queens*, and *kalah*) for which the serial elision was not easily computable due to the use of Cilk inlets. Furthermore, we left out *hello* as it is a trivial hello world program, and *nfib* as it is basically the same as *fib*. For each of the remaining applications we measured the overall program speedup as not all applications come with their own measurement of relevant program parts. While it seems to be the most appropriate way to us to treat all benchmarks of the suite in the same way,

[5]Earlier results similar to those shown in Figure 8(b) have been reported in our own earlier work [Streit et al. 2013].

measuring whole program speedup results in lower speedups than one might wish to see on a quad-core machine. In most cases this is caused by large parts of the application being inherently sequential: for instance, allocating and filling large arrays to sort an finally verifying result correctness. For *cilksort*, *matmul*, and *spacemul* the difference is quite large: While their measured kernels have been accelerated by factors of *4.32*, *3.63*, and *3.26*, respectively, the overall speedup is significantly lower.

Note that we report three numbers for *fft* that all represent the same program but run on different inputs,[6] each covering a different characteristical execution path through the application.

Overall, the measurements confirm our hypotheses:

(1) *Sambamba* subsumes different parallelization approaches by effectively detecting and leveraging different forms of parallelism.
(2) Parallelization enabling techniques like privatization and reduction recognition are used where applicable.
(3) The runtime is comparable to state-of-the-art parallelization tools, but no developer guidance is needed.

7. RELATED WORK

7.1. Reduction

The usual definitions of reduction, for example that of Midkiff [2012], are based more or less on the syntactic form of a reduction statement and miss important opportunities. The *extended reduction statements* of Rauchwerger and Padua [1995] share properties with ours, but have no notion of varying reduction locations. Dynamic approaches like *Privateer* by Johnson et al. [2012] or the *LRPD-test* by Rauchwerger and Padua [1995] try to avoid reliance on statically analyzable reductions of a particular syntactic form. These approaches optimistically assume promising accesses to be reductions and parallelize the containing loops at the cost of having to perform runtime checks to validate the decisions made.

The approach of reduction recognition described in this article is not as restrictive as the usual definitions based on syntactic features. Nevertheless, it is by definition a static approach and consequently not able to detect all possible reductions.

7.2. Static Parallelization

Burke et al. [1989] describe the exploitation of nested fork-join parallelism while taking into account the possibility of privatization. The approach does not trade parallelism for overhead and parallelizes everything it can. The motivation of the approach described in this article is similar to ours and we are certainly inspired by the work of Burke et al.

In a similar fashion, Sarkar [1991] presents a heuristic-based approach to statically parallelize task trees computed from the program dependence graph of FORTRAN functions. This enforced tree structure, motivated by the requirement to generate a parallel FORTRAN program with structured parallelism, limits the flexibility of the approach. Linear programming based scheduling of hierarchical task graphs for embedded systems by Cordes et al. [2010] shares this limitation and further imposes restrictions on the shape of the generated parallel code regions.

Rugina and Rinard [1999] propose a simple method to automatically parallelize divide-and-conquer algorithms by spanning parallel sections around call sites that are assumed to be spawned off for parallel work by introducing *Cilk spawn* and *sync*

[6] *fft* inputs: “10,000,000”, “33,554,432” and “-c”.

primitives. This approach is very limited and unable to abstract from the implemented control flow.

Decoupled Software Pipelining (DSWP) aims at parallelizing sequential loops by forming patterns of pipelined execution [Rangan et al. 2004]. Loops are decomposed into pipeline stages, possibly executing in parallel to each other. Each stage communicates produced values to the threads executing later stages as needed. DSWP has been extended by automatically performing the thread extraction [Ottoni et al. 2005], by allowing one to distribute a single pipeline stage to different threads [Raman et al. 2008], by allowing one to speculatively parallelize [Vachharajani et al. 2007], and by enabling cross-invocation parallelism among different loop instances [Huang et al. 2013] for loops of a specific shape. Huang et al. [2010] generalize the idea of Raman et al. [2008] and enable the parallelization of individual DSWP stages by manually applying a secondary loop parallelization scheme. The work clearly shows that different parallelization schemes can be profitably combined. However, the question on how to automatically select and prioritize different approaches is considered to be a challenging open research question by the authors. While modern implementations of DSWP, like *Parcae* [Raman et al. 2012] for instance, avoid it, the earlier approaches rely on specialized hardware for interthread communication and recovery from mis-speculation. *Sambamba* instead runs on commodity systems.

Vandierendonck et al. [2010] describes *Paralax*, a semiautomatic approach of parallelization in a DSWP-like fashion. The approach relies on DSA for its dependence analysis, and suffers from the same imprecisions as we do. To address this concern, Vandierendonck et al. [2010] motivates a set of user annotations.

Zhong et al. [2008] describe an approach of automatic speculative DOALL parallelization of loops relying on hardware transactional memory, hardware-based low-cost thread spawning, and low-latency intercore communication. Mehrara et al. [2009] implement a software transactional memory system to get rid of these hardware requirements. The described STM is specialized and limited to automatic DOALL parallelization of loops. Kim et al. [2012] apply speculative DOALL parallelization to distribute the computation performed by a loop to a cluster of machines.

In Helix [Campanoni et al. 2012], loop iterations are automatically distributed in a round-robin fashion to different threads. The latency of necessary intercore communication is hidden by exploiting the SMT capabilities of modern multicore processors. While the performance results are impressive, the authors show in their own follow-up work [Campanoni et al. 2014] that the approach does not scale to more than four cores and propose hardware support to overcome this limitation. Both approaches are limited to parallel execution of a single loop at a time.

7.3. Runtime Parallelization

Kulkarni et al. [2007] require the programmer to use the graph data structures and iterators provided by their Galois system in order to make dependencies between graph nodes explicit. In return, these explicitly stated dependencies enable dynamic parallel execution of graph-based algorithms without the need for conservative assumptions.

Out of order Java by Jenista et al. [2011] and improved by Eom et al. [2012] provide a task extension to the Java language allowing the programmer to mark regions of the code to be considered for parallel execution. The compiler generates lightweight runtime checks, enabling efficient prevalidation of potential conflicts at runtime before spawning a parallel task. Both approaches rely on the programmer to rethink and rewrite the subject application. Chen and Olukotun [2003] implemented a runtime system for Java applications that dynamically monitors dependencies between loop iterations. To find promising parallelizable loops, the approach relies on a hardware profiler. DeVuyst et al. [2011] and Hertzberg and Olukotun [2011] followed a similar

idea. By employing runtime binary translation, their approaches do not rely on the availability of the application source code. Johnson et al. [2007] makes heavy use of thread level speculation supporting hardware to empirically optimize an application after a profiling run preceding the actual execution. All these approaches rely on special hardware in contrast to the work presented in this article.

The *Parcae* system by Raman et al. [2012] provides a flexible parallel execution environment and promises to allow for holistic optimization of a parallel program instead of mere parameter tuning as, for example, done by Karcher and Pankratius [2011] in the context of parallelization. Parcae relies on extensions to the operating system to orchestrate the parallel execution of different applications.

8. LIMITATIONS AND FUTURE WORK

While we are convinced that the described approach is an important step towards unifying several important parallelization approaches, we are not there yet. Our implementation of *Generalized Task Parallelism* has several limitations, mostly of technical nature. Nevertheless, these limitations keep our implementation from parallelizing code that it could handle in principle.

In this section we will give a nonexhaustive list of the most relevant limitations and a short hint on how we plan to address them in the future. The purpose is to make clear, that we are aware of technical limitations, which should not be understood as limitations of the general approach.

Flow Insensitivity. The fact that the data-structure analysis we use [Lattner et al. 2007] is flow-insensitive causes imprecision when trying to identify memory regions as disjoint. This behavior can of course be avoided by employing a flow-sensitive analysis. The problem is that flow- and context-sensitive analyses usually do not scale very well. We plan to address this issue by using a staged approach as proposed by Hardekopf and Lin [2011], or a client-driven one as, for example, that of Guyer and Lin [2005].

Dependence Analysis. With the goal of parallelizing general-purpose applications we chose to use DSA, which is a points-to analysis that is particularly well suited for irregular data structures. However, this analysis has two major weak spots relevant in our situation: It is not able to precisely deal with regular data structures like arrays, and it greatly overapproximates the effects of recursive functions.

We are currently working on a new approach combining and extending ideas of the range analysis by Rugina and Rinard [1999], and the runtime parametric memory access analysis of Rus et al. [2003].

Apart from general scalability and stability improvements of the implementation, we plan to further extend the approach to make use of runtime information and dynamic adaptation.

9. CONCLUSION

In this article, we presented *Sambamba*, an approach to naturally unify different forms of loop parallelization as well as fork-join-style task parallelization, reduction, privatization, and speculation. We express the freedom to choose from all these alternatives in an integer linear programming approach to PDG scheduling that considers all parallelization opportunities at once and find the optimal trade-off with respect to a given cost function.

Facing the diversity and complexity of modern processors, memory systems, runtime environments, and application inputs, no static approach will ever be able to predict the profitability of a particular parallel code version. Therefore, the described approach

relies on an adaptive runtime system to continuously recombine and reassess parallelization decisions and to adapt to changing requirements.

We validated experimentally that *Sambamba* detects and exploits parallelism in a variety of programs from many different benchmark suites exhibiting different kinds of parallelism. We consistently achieve speedups at the same level or better than state-of-the-art parallelizing tools, or manual parallelization.

REFERENCES

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. ACM, New York, NY, 72–81. DOI:http://dx.doi.org/10.1145/1454115.1454128

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.* 30, 8 (Aug. 1995), 207–216. DOI:http://dx.doi.org/10.1145/209937.209958

Michael Burke, Ron Cytron, Jeanne Ferrante, and Wilson Hsieh. 1989. Automatic generation of nested, fork-join parallelism. *J. Supercomput.* 3, 2 (1989), 71–88. DOI:http://dx.doi.org/10.1007/BF00129843

Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David Brooks. 2014. HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs. *SIGARCH Comput. Archit. News* 42, 3 (June 2014), 217–228. DOI:http://dx.doi.org/10.1145/2678373.2665705

Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. 2012. HELIX: Automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO'12)*. ACM, New York, NY, 84–93. DOI:http://dx.doi.org/10.1145/2259016.2259028

Michael K. Chen and Kunle Olukotun. 2003. The jrpm system for dynamically parallelizing java programs. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*. ACM Press, New York, NY, 434–446. DOI:http://dx.doi.org/10.1145/859618.859668

Daniel Cordes, Peter Marwedel, and Arindam Mallik. 2010. Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES/ISSS'10)*. ACM, New York, NY, 267–276. DOI:http://dx.doi.org/10.1145/1878961.1879009

Matthew DeVuyst, Dean M. Tullsen, and Seon Wook Kim. 2011. Runtime parallelization of legacy code on a transactional memory system. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC'11)*. ACM, New York, NY, 127–136. DOI:http://dx.doi.org/10.1145/1944862.1944882

Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. 2009. Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *Proceedings of the 2009 International Conference on Parallel Processing (ICPP'09)*. IEEE Computer Society, Washington, DC, 124–131. DOI:http://dx.doi.org/10.1109/ICPP.2009.64

Tobias Edler von Koch and Björn Franke. 2014. Variability of data dependences and control flow. In *Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14)*. 180–189. DOI:http://dx.doi.org/10.1109/ISPASS.2014.6844482

Yong hun Eom, Stephen Yang, James C. Jenista, and Brian Demsky. 2012. DOJ: Dynamically parallelizing object-oriented programs. *SIGPLAN Not.* 47, 8 (Feb. 2012), 85–96. DOI:http://dx.doi.org/10.1145/2370036.2145828

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. DOI:http://dx.doi.org/10.1145/24039.24041

Samuel Z. Guyer and Calvin Lin. 2005. Error checking with client-driven pointer analysis. *Sci. Comput. Prog.* 58, 1–2 (Oct. 2005), 83–114. DOI:http://dx.doi.org/10.1016/j.scico.2005.02.005

Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'11)*. IEEE Computer Society, Washington, DC, 289–298.

Ben Hertzberg and Kunle Olukotun. 2011. Runtime automatic speculative parallelization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'11)*. IEEE Computer Society, Washington, DC, 64–73.

Jialu Huang, Stephen R. Beard, Nick P. Johnson, Thomas B. Jablin, and David I. August. 2013. Automatically exploiting cross-invocation parallelism using runtime information. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'13)*. IEEE Computer Society, Washington, DC, USA, 1–11. DOI:http://dx.doi.org/10.1109/CGO.2013.6495001

Jialu Huang, Arun Raman, Thomas B. Jablin, Yun Zhang, Tzu-Han Hung, and David I. August. 2010. Decoupled software pipelining creates parallelization opportunities. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10)*. ACM, New York, NY, 121–130. DOI:http://dx.doi.org/10.1145/1772954.1772973

James Christopher Jenista, Yong hun Eom, and Brian Charles Demsky. 2011. OoOJava: Software out-of-order execution. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*. ACM, New York, NY, 57–68. DOI:http://dx.doi.org/10.1145/1941553.1941563

Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August. 2012. Speculative separation for privatization and reductions. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*. ACM, New York, NY, 359–370. DOI:http://dx.doi.org/10.1145/2254064.2254107

Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. 2007. Speculative thread decomposition through empirical optimization. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*. ACM, New York, NY, 205–214. DOI:http://dx.doi.org/10.1145/1229428.1229474

Thomas Karcher and Victor Pankratius. 2011. Run-Time automatic performance tuning for multicore applications. In *Euro-Par 2011 Parallel Processing*, Emmanuel Jeannot, Raymond Namyst, and Jean Roman (Eds.). Lecture Notes in Computer Science, Vol. 6852. Springer, Berlin, 3–14. DOI:http://dx.doi.org/10.1007/978-3-642-23400-2_2

Ken Kennedy and John R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, San Francisco, CA.

Hanjun Kim, Nick P. Johnson, Jae W. Lee, Scott A. Mahlke, and David I. August. 2012. Automatic speculative DOALL for clusters. In *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO'12)*. ACM, New York, NY, 94–103. DOI:http://dx.doi.org/10.1145/2259016.2259029

Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, New York, NY, 211–222. DOI:http://dx.doi.org/10.1145/1250734.1250759

Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. *SIGPLAN Not.* 42, 6 (June 2007), 278–289. DOI:http://dx.doi.org/10.1145/1273442.1250766

Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. 2009. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. *SIGPLAN Not.* 44, 6 (June 2009), 166–176. DOI:http://dx.doi.org/10.1145/1543135.1542495

Samuel P. Midkiff. 2012. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Morgan & Claypool Publishers.

Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. 2005. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 38)*. IEEE Computer Society, Washington, DC, 105–118. DOI:http://dx.doi.org/10.1109/MICRO.2005.13

Louis-Noël Pouchet. 2012. PolyBench/C: The Polyhedral Benchmark Suite. Retrieved from http://www.cs.ucla.edu/~pouchet/software/polybench/.

Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. 2012. Parcae: A system for flexible parallel execution. *SIGPLAN Not.* 47, 6 (June 2012), 133–144. DOI:http://dx.doi.org/10.1145/2345156.2254082

Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. 2008. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'08)*. ACM, New York, NY, 114–123. DOI:http://dx.doi.org/10.1145/1356058.1356074

Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. 2004. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*. IEEE Computer Society, Washington, DC, 177–188. DOI:http://dx.doi.org/10.1109/PACT.2004.14

Lawrence Rauchwerger and David Padua. 1995. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI'95)*. ACM, New York, NY, 218–232. DOI:http://dx.doi.org/10.1145/207110.207148

Radu Rugina and Martin Rinard. 1999. Automatic parallelization of divide and conquer algorithms. *SIGPLAN Not.* 34, 8 (May 1999), 72–83. DOI:http://dx.doi.org/10.1145/329366.301111

Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. 2003. Hybrid Analysis: Static and dynamic memory reference analysis. *Int. J. Parallel Program.* 31, 4 (Aug. 2003), 251–283. DOI:http://dx.doi.org/10.1023/A:1024597010150

Vivek Sarkar. 1991. Automatic partitioning of a program dependence graph into parallel tasks. *IBM J. Res. Dev.* 35, 5–6 (Sept. 1991), 779–804. DOI:http://dx.doi.org/10.1147/rd.355.0779

Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. 2013. Sambamba: Runtime adaptive parallel execution. In *Proceedings of the 3rd International Workshop on Adaptive Self-Tuning Computing Systems (ADAPT'13)*. ACM, New York, NY, Article 7, 6 pages. DOI:http://dx.doi.org/10.1145/2484904.2484911

Sid Ahmed Ali Touati, Julien Worms, and Sébastien Briais. 2013. The Speedup-Test: A statistical methodology for programme speedup analysis and computation. *Concurr. Comp.: Pract. Exper.* 25, 10 (2013), 1410–1426. DOI:http://dx.doi.org/10.1002/cpe.2939

Peng Tu and David A. Padua. 1994. Automatic array privatization. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, London, UK, 500–521. http://dl.acm.org/citation.cfm?id=645671.665384

Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. 2007. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT'07)*. IEEE Computer Society, Washington, DC, 49–59. DOI:http://dx.doi.org/10.1109/PACT.2007.66

Hans Vandierendonck, Sean Rul, and Koen De Bosschere. 2010. The paralax infrastructure: Automatic parallelization with a helping hand. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. ACM, New York, NY, 389–400. DOI:http://dx.doi.org/10.1145/1854273.1854322

Hongtao Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. 2008. Uncovering hidden loop level parallelism in sequential applications. In *Proceedings of the IEEE 14th International Symposium on High Performance Computer Architecture (HPCA'08)*. 290–301. DOI:http://dx.doi.org/10.1109/HPCA.2008.4658647

Received June 2014; revised December 2014; accepted January 2015