Bamboo: A Data-Centric, Object-Oriented Approach to Many-core Software

Jin Zhou   Brian Demsky

Department of Electrical Engineering and Computer Science
University of California, Irvine
Irvine, CA 92697
{jzhou1,bdemsky}@uci.edu

Abstract

Traditional data-oriented programming languages such as dataflow languages and stream languages provide a natural abstraction for parallel programming. In these languages, a developer focuses on the flow of data through the computation and these systems free the developer from the complexities of low-level, thread-oriented concurrency primitives. This simplification comes at a cost — traditional data-oriented approaches restrict the mutation of state and, in practice, the types of data structures a program can effectively use. Bamboo borrows from work in typestate and software transactions to relax the traditional restrictions of data-oriented programming models to support mutation of arbitrary data structures.

We have implemented a compiler for Bamboo which generates code for the TILEPro64 many-core processor. We have evaluated this implementation on six benchmarks: Tracking, a feature tracking algorithm from computer vision; KMeans, a K-means clustering algorithm; MonteCarlo, a Monte Carlo simulation; FilterBank, a multi-channel filter bank; Fractal, a Mandelbrot set computation; and Series, a Fourier series computation. We found that our compiler generated implementations that obtained speedups ranging from $26.2\times$ to $61.6\times$ when executed on 62 cores.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; D.3.2 [*Language Classifications*]: Data-flow languages; G.1.6 [*Optimization*]: Gradient Methods

General Terms Algorithms, Languages

Keywords Many-core Programming, Data-Centric Languages

1. Introduction

With the wide-scale deployment of multi-core processors and the impending arrival of many-core processors, software developers must write parallel software to realize the benefits of continued improvements in microprocessors. Developing parallel software using today's development tools can be challenging. These tools require developers to expose parallelism as threads and then control concurrent access to data with locks. Reasoning about the correctness and performance of these systems has proven to be difficult.

In the past, mainstream processors have presented software developers with a relatively static programming target at the language abstraction level. We expect this will no longer be true — in the future as fabrication technologies advance, the number and types of cores in each successive microprocessor generation will change. Adapting thread-based applications to these changes may require significant refactoring efforts. Developing software for many-core processors will clearly benefit from both new tools and languages.

This paper presents a data-oriented extension to Java. In this approach, the developer focuses on how data flows through the application. Our approach borrows inspiration from dataflow and stream-based languages and extends the basic data-oriented approach for use in imperative, object-oriented languages. Traditional data-oriented programming approaches place severe restrictions on how programs mutate data structures; Bamboo relaxes these restrictions to allow programs to freely mutate data structures.

Bamboo programs are composed of a set of tasks that implement the program's operations. Objects have abstract states associate with them and tasks contain parameter guards that specify the abstract states of the parameter objects that they can operate on. Tasks have data-oriented invocation semantics: when there exist objects with abstract states that satisfy a task's guards, the runtime invokes the task on the objects. Tasks can in turn modify the abstract states of the objects they operate on.

Bamboo's abstract object states are similar to typestates [31]. Like typestates, operations can change the abstract states of objects and an object's abstract state determines which operations can be invoked on the object. However, Bamboo's abstract object states differ from typestates in that abstract object states are dynamically computed and used to determine which operation to invoke next, while typestates are used to statically detect errors in object usage.

Conceptually, abstract states serve to control access to data at the task level. For example, an email client may have message objects with both an edit state and a sent state. The send task might transition a message object into the sent state, preventing any further edits to the message.

Bamboo supports traditional method calls from both tasks and other methods. Bamboo restricts the usage of global variables — tasks (and methods) can only read their parameter objects or objects reachable from the parameters. Tasks do not maintain persistent state between invocations — therefore, a Bamboo implementation can safely create multiple parallel instantiations of a task if these instantiations operate on disjoint parameter objects.

The Bamboo compiler is staged as follows:

- **Dependence Analysis:** The compiler performs a static dependence analysis to determine (1) the set of abstract states that objects can reach and (2) how tasks cause objects to transition through these abstract states. The analysis generates as output a finite state machine for each class in which the states model the abstract object states and the transitions model the effects of tasks on these abstract states.
- **Disjointness Analysis:** The compiler performs a static disjointness analysis on the Java-like imperative code inside Bamboo tasks and methods to determine whether a task introduces aliasing between parameter objects. Bamboo uses the disjointness analysis to automatically generate fine-grained locking code that ensures transactional task semantics. Bamboo transactions are light-weight — at invocation, a task simply locks its parameter objects. If the runtime cannot lock all of a task's parameters, the runtime releases the locks and executes a different task. Tasks never abort and they incur no extra overheads.
- **Implementation Generation Algorithm:** The implementation generation algorithm takes as input the static analyses results and profile information. The algorithm uses a set of transformations to generate many candidate many-core implementations of the application to serve as starting points for optimization.
- **Implementation Optimization:** The evaluation stage uses high-level abstract simulation to evaluate the relative performance of the implementations. The compiler performs a critical path analysis on the simulated execution to identify opportunities to improve the implementation and uses the results to direct a simulated annealing-based search. The compiler generates code for the best implementation.

1.1 Relation with Stream, Dataflow, and Tuple-space Languages

Bamboo is closely related to both stream and dataflow languages. Stream and dataflow languages traditionally impose severe restrictions on how an application can mutate data and often forbidding it. They often require applications to access data structures in deterministic patterns. These restrictions are imposed because these languages typically do not contain concurrency primitives that restrict access to shared state and instead must make the state immutable.

Bamboo combines transactional task semantics together with abstract object states to control access to shared data structures. This enables Bamboo to support mutation of structurally complex shared data structures in a data-oriented programming model. Bamboo's task parameter objects are intended to be the roots of disjoint heap data structures. The Bamboo compiler includes a static disjointness analysis that detects violations of this disjointness property and generates a locking strategy that guarantees transactional semantics. The disjointness analysis enables the Bamboo implementation to efficiently provide transactional task semantics.

The combination of abstract object states and task dispatch allows Bamboo to support applications that must access shared data structures at non-deterministic times in the computation. Abstract object states serve to control these accesses.

Tuple-space languages contain similar constructs to Bamboo's global object space. However, threads in tuple-space applications can contain internal state and can manipulate the tuple-space in arbitrary ways. These differences make it difficult for a compiler to automatically understand the role of a thread in a tuple-space language and frustrate efforts to automatically parallelize the thread by creating multiple instantiations.

1.2 Contributions

This paper makes the following contributions:

- **Hybrid Data-Oriented Approach:** It presents the Bamboo language, which implements a data-oriented programming model in the context of an imperative, object-oriented programming language.
- **Data Structure Mutation:** It presents a dataflow-based programming model that supports data structure mutation. Many algorithms are most naturally expressed in terms of state mutations. This approach allows developers to easily code these algorithms while still benefiting from high-level language support for dataflow-type constructs.
- **Automatic Implementation Optimization:** It presents an algorithm that automatically generates many candidate many-core implementations of the application. This algorithm combines profile information with a simulation-based implementation evaluation algorithm to generate many-core implementations that are optimized for the target processor.
- **Evaluation:** It presents an evaluation of Bamboo on a 64-core TILEPro64 microprocessor [2] for several benchmarks. The TILEPro64 processor is a many-core CPU and representative of the many-core microprocessors that will become commonplace in the future. We found that it was relatively simple to develop Bamboo implementations of the benchmarks and that the implementations made effective use of available cores. The experimental results show that Bamboo is able to achieve speedups between $26.2\times$ and $61.6\times$ for our benchmarks. Moreover, we found that Bamboo successfully generated a sophisticated heterogeneous implementation of the MonteCarlo benchmark that used pipelining to overlap the simulation and aggregation tasks.

The remainder of the paper is structured as follows. Section 2 presents an example that we use to illustrate our approach. Section 3 presents the Bamboo language. Section 4 presents the implementation synthesis algorithm. Section 5 presents our evaluation of the approach on several benchmark applications. Section 6 discusses related work; we conclude in Section 7.

2. Example

We present a keyword counting example to illustrate Bamboo.

2.1 Classes

Figure 1 presents part of the `Text` class declaration for the example. The keyword counter uses instances of the `Text` class to divide the input text into sections and count occurrences of each word.

Class declarations contain declarations for the class's abstract states. An abstract state is declared with the keyword `flag` followed by a name. Bamboo's abstract states support orthogonal classifications of objects: an object may simultaneously be in more than one abstract state. The runtime uses the abstract state of an object to determine which *tasks* to invoke on the given object. When a task exits, it can change the values of the flags of its parameter objects.

```
class Text {
  flag process;
  flag submit;
  ...
}
```

Figure 1. Text Class Declaration

The `Text` class in the example contains two abstract state declarations: the `process` flag, which indicates that the `Text` object is ready to be processed, and the `submit` flag, which indicates that the `Text` object can submit its result.

2.2 Tasks

Bamboo applications are structured as a collection of tasks. The key difference between tasks and methods is that tasks have data-oriented invocation semantics: the runtime invokes a task when the heap contains objects with the appropriate abstract state to serve as the task's parameters. Note that while the runtime controls task invocation, tasks can call methods. The runtime uses a task's specification to determine which objects serve as the task's parameters and when to invoke the task.

Each task declaration consists of the keyword `task`, the task's name, the task's parameters, and the body of the task. Figure 2 presents the task declarations for the example. The first task declaration declares the `startup` task. The guard `in initialstate` declares that the `StartupObject` object must have its `initialstate` flag set before the runtime can invoke this task. The runtime invokes the task when there exist parameter objects in the heap that satisfy the parameters' guard expressions. Before exiting, the `taskexit` statement in the `startup` task resets the `initialstate` flag in the `StartupObject` to false to prevent the runtime from repeatedly invoking the `startup` task.

The abstract state-based programming model is a powerful construct for specifying which objects a task operates on. For many applications, it is possible to specify these dependencies with graphical task dependence diagrams. Bamboo supports the use of external graphical tools in which a developer would draw a dependence diagram from which the tool automatically generates task declarations.

```
task startup(StartupObject s in initialstate) {
  Partitioner p = new Partitioner(s.args[0]);
  while(p.morePartitions()) {
    String section = p.nextpartition();
    Text tp = new Text(section) {process:=true};
  }
  Results rp =
    new Results(p.sectionNum()) {finished:=false};
  taskexit(s: initialstate:=false);
}
task processText(Text tp in process) {
  tp.process();
  taskexit(tp: process:=false, submit:=true);
}
task mergeIntermediateResult(
  Results rp in !finished, Text tp in submit) {
  boolean allprocessed = rp.mergeResult(tp);
  if (allprocessed)
    taskexit(rp: finished:=true;
             tp: submit:=false);
  taskexit(tp: submit:=false);
}
```

Figure 2. Flag Specifications for Tasks

2.3 Execution

We next describe the execution of the example. It performs the following operations (although not necessarily in this order):

- **Startup:** The runtime creates a `StartupObject` object in the `initialstate` abstract state. This causes the runtime to invoke the `startup` task. This task creates a `Partitioner` object to partition the text stream into sections. For each section, the task creates a new `Text` object in the `process` abstract state, which indicates that the object is ready for processing. The task then creates a `Results` object to merge the intermediate results. Finally, it transitions the `StartupObject` object out of the `initialstate` abstract state to prevent the runtime from repeatedly invoking the `startup` task.
- **Processing a Text Section:** When the runtime identifies a `Text` object in the `process` abstract state, it invokes the `processText` task on that object to process the text section and stores the intermediate result in the object. Upon exiting, the object is transitioned from the `process` abstract state to the `submit` abstract state to indicate that the intermediate result can be merged into the final result and to prevent repeated invocations of `processText` task on this object.
- **Merging Results:** The `mergeIntermediateResult` task merges the intermediate results from the `Text` objects into the `Results` object. It transitions the `Text` object out of the `submit` abstract state to prevent merging the same object again. If it has merged all of the intermediate results, it transitions the `Results` object to the `finished` abstract state.

2.4 Scheduling

The Bamboo compiler uses profile information and a processor description to generate a binary that is optimized for both the application's runtime behavior and the target processor. The Bamboo compiler contains an automatic implementation synthesis and evaluation-based optimization framework that generates and evaluates many possible candidate implementations of the application to synthesize an optimized many-core implementation.

The Bamboo compiler generates a *combined state transition graph* (CSTG) to reason about the possible behaviors of the application. Figure 3 presents the CSTG for the example. The nodes in this graph model the abstract object states for the classes that serve as task parameters. Two concentric ellipses indicate that the object can be allocated with this abstract state. Solid edges model the state transitions caused by the invocation of a task on an object. Dashed edges model the creation of new objects — a *new object edge* points from the task that creates an object to the abstract state node that abstracts the state of the newly created object.

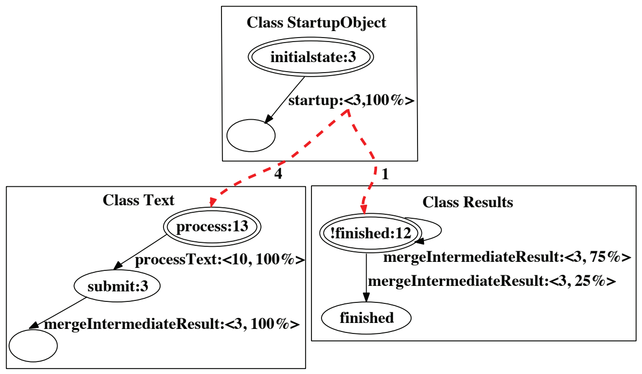


Figure 3. CSTG for the Keyword Counting Example

The compiler associates profile information with the nodes and edges in the CSTG. The CSTG combined with the profile information forms a Markov model [26] of the program's execution. The solid edges contain labels with the name of a task followed by a colon and a tuple that contains the expected time the task takes to execute if it makes this transition and the probability that the task will take this transition. The dashed edges are labeled with a tuple that gives the expected number of newly created objects. For example, the edge in Figure 3 from the `startup` task to the `process` state is labeled `4` indicating that the task may generate four new `Text` objects in the `process` state. Each node contains an abstract object state followed by a colon and a lower bound estimate of the time it takes to complete processing an object in this state.

The compiler transforms the CSTG to optimize the application's implementation for a given processor. For example, the algorithm might begin by generating multiple instantiations of the tasks that process `Text` objects. The compiler uses these transformations to generate candidate layouts (as presented in Figure 4) to serve as starting points for the directed-simulated annealing optimization algorithm. The algorithm first runs a high-level simulation to evaluate these layouts using the profile statistics. Then it computes the critical path subject to scheduling constraints of the simulated execution and uses this critical path along with dependence information to identify a set of tasks that can potentially be migrated to different cores to reduce the length of the critical path. It then generates a set of candidate layouts that implement these transformations as the starting point for an iterative optimization procedure. When it reaches the point of diminishing returns, the compiler selects the best candidate layout and generates the corresponding executable.

Figure 4 presents a candidate layout of the keyword counting example for a quad core processor. This implementation deploys all tasks on core 0 while deploying only the `processText` task on the other three cores. The execution distributes the `Text` objects to all 4 cores in a round-robin fashion.

3. Bamboo Language

Bamboo applications are composed of a set of tasks that implement high-level operations and a set of task declarations that describe when to execute these tasks. Bamboo tasks are written in an object-oriented, type-safe subset of Java. The task declaration language describes the data dependencies between tasks.

Tasks are blocks of code that encapsulate individual conceptual operations. Each task contains a task declaration that the runtime uses to determine (1) when to execute the task, (2) what data the task needs, and (3) how the task changes the role this data plays in the computation. Thus the set of task declarations describes the dependencies between the tasks. Bamboo associates abstract object states with objects. The abstract object states are used to determine which tasks should be invoked on an object.

Figure 5 presents the grammar for Bamboo's task extensions to Java. Each task contains a set of guards that specify when the runtime should invoke the task. The guards contain a set of predicates on the abstract states of the parameter objects. Only objects whose abstract state satisfy the task's guards can serve as a parameter object to the task. Tasks can in turn modify an object's abstract state when (1) the object is allocated or (2) at the completion of its execution.

An abstract object state is declared in a class declaration by using the `flag` keyword followed by the state's name. Developers can use abstract object states to capture orthogonal aspects of an object's current role in the computation: therefore an object can simultaneously be in more than one abstract object state.

The runtime uses an object's abstract state to determine which tasks to invoke on that object. It uses a task's declaration to determine which objects can serve as the task's parameters and when to invoke the task. When the heap contains parameter objects with the specified abstract states to serve as a task's parameters, the runtime invokes that task on those objects.

Bamboo provides a *tag* construct to support reuse of blocks of tasks and to group related objects. Consider a Bamboo task-based library routine[1] that takes as input an `Image` object in the `uncompressed` state, compresses the image, and then transitions

[1] Bamboo also supports standard method-based libraries.

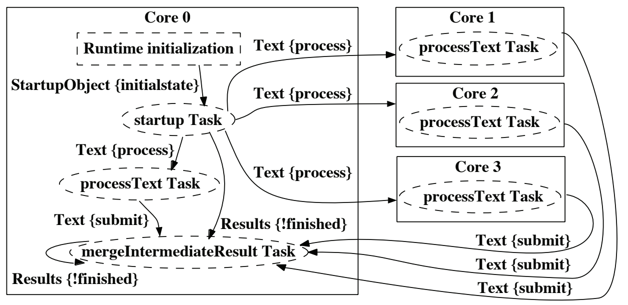


Figure 4. Scheduling Layout on a Quad Core Processor

flagdecl	:=	`flag` *flagname*;
tagdecl	:=	*tagtype tagname*;
taskdecl	:=	`task` *name*(*taskparamlist*)
taskparamlist	:=	*taskparamlist*, *taskparam* | *taskparam*
taskparam	:=	*type name* `in` *flagexp* `with` *tagexp* |
		type name `in` *flagexp*
flagexp	:=	*flagexp* `and` *flagexp* | *flagexp* `or` *flagexp* |
		!*flagexp* | (*flagexp*) | *flagname* | `true` | `false`
tagexp	:=	*tagexp* `and` *tagtype tagname* | *tagtype tagname*
statements	:=	... | `taskexit`(*flagactionlist*) |
		`tag` *tagname* = `new tag`(*tagtype*) |
		`new` *name*(*params*){*flagortagactions*}
flagactionlist	:=	*flagactionlist*; *name* : *flagortagactions* |
		name : *flagortagactions*
params	:=	... | `tag` *tagname*
flagortagactions	:=	*flagortagactions*, *flagortagaction* |
		flagortagaction
flagortagaction	:=	*flagaction* | *tagaction*
flagaction	:=	*flagname* := *boolliteral*
tagaction	:=	`add` *tagname* | `clear` *tagname*

Figure 5. Task Grammar

the `Image` object to the `compressed` state. A `startsave` task for a graphics editing application might take as input a `Drawing` object and create an `uncompressed Image` object. A block of tasks in the library would then perform a set of task invocations that eventually transition the `uncompressed Image` object to the `compressed` state. A second `finishsave` task would then take as input the `Drawing` object and the `compressed Image` object. It is important that the `finishsave` task receives the `compressed Image` for the specific `Image` object the `startsave` task created and not some other `Image` object. Tags solve this problem. The `startsave` task creates a tag and tags both the `Image` object and the `Drawing` object. The `finishsave` task declaration then specifies that the `Drawing` object and the `Image` object both have the same tag, ensuring that it receives the correct `Image` object.

Tags serve a second purpose. In addition to disambiguating different uses of the same object, they can disambiguate different instances of the same use. Suppose that two `Drawing` objects were simultaneously saved. Without tags, it is possible for a `compressed Image` object to be associated with the wrong `Drawing` object. Tags are allocated using the `new tag` statement. Methods can declare `tag` parameters and tag instances can be passed into a method call.

Bamboo provides a modified `new` object allocation statement. This statement takes as input the initial abstract state for objects allocated at this site and a list of tag variables whose tag instances should be bound to the newly allocated objects. Bamboo provides the `taskexit` statement that can change the abstract states and tag bindings of the task's parameter objects and then exits the task.

The current implementation of Bamboo is data-oriented at the top-level — Bamboo does not contain threads and Bamboo applications are started by the creation of a `StartupObject` object. It is straightforward to implement Bamboo as a strict extension to Java — thread-based code would then use data-oriented components through an asynchronous calling interface.

4. Implementation Synthesis

We next present the Bamboo compiler's algorithm for generating an optimized implementation. The Bamboo compiler uses the following staged strategy to synthesize application implementations:

1. **Dependence Analysis:** The dependence analysis processes the task declarations to characterize the tasks' data dependences.
2. **Disjointness Analysis:** Disjointness analysis processes the imperative code inside of Bamboo tasks and methods to determine whether it introduces sharing between different task parameter objects. The compiler uses this information to generate locks that guarantee transactional semantics for task invocation.
3. **Candidate Implementation Generation:** Candidate implementation generation synthesizes non-isomorphic candidate implementations. Several implementations are randomly generated to serve as a starting point for further optimization.
4. **Simulation-based Evaluation:** The evaluation phase uses profile information along with an architectural specification to perform a high-level simulation of the candidate implementation. If the profile indicates that the application terminates, the simulation computes an estimated execution time. Otherwise, it estimates the percentage of the time spent doing useful work.
5. **Optimization:** Based on the evaluation results, the final stage uses directed-simulated annealing to iteratively optimize the candidate implementations to improve their performance.

We next present each of these stages in more detail.

4.1 Dependence Analysis

The dependence analysis operates on *abstract state transition graphs* (ASTGs) [17]. An ASTG is associated with an object type and abstracts the possible state transitions of instances of that type. An ASTG is composed of *abstract state nodes* and edges between these nodes. An abstract state node represents the abstract state and tag components of an object's state — it contains the states of all the object's abstract states and a 1-limited count (0, 1, or at least 1) of the tag instances of each type that are bound to the object. If an object in the computation can reach a given abstract state, the abstract state transition graph for that object's class contains the corresponding abstract state node. The edges in the ASTG abstract the actions of tasks on objects. If a task can transition an object from one abstract state to a second abstract state, then there is an edge labeled with that task from the abstract state node that corresponds to the first abstract state to the abstract state node that corresponds to the second abstract state.

4.2 Disjointness Analysis

Disjointness analysis determines whether the parts of the heap reachable from distinct task parameter objects are disjoint [23]. Disjointness analysis differs from pointer analysis in that it can determine that two objects represented by the same static node are distinct if the parameter objects they are reachable from are distinct. The analysis reasons about static reachability graphs, which characterize the reachability of each object in the heap from a select set of root objects. Nodes in reachability graphs represent objects and edges represent heap references. The graphs are annotated with sets of reachability states that describe which objects can reach other objects. The analysis uses the reachability states to determine if a task introduces sharing between the parts of the heap reachable from two different parameter objects. If the analysis determines that a task may create sharing between the disjoint heap regions associated with two different parameter objects, the compiler generates code that adds a shared lock for the two parameter objects.

4.3 Candidate Implementation Generation

This stage in the compilation process generates several candidate implementations. These candidates serve as a starting point for later evaluation and optimization stages. The implementation generation process is structured as three steps: (1) generate a combined state transition graph that characterizes the dependences between parts of the application, (2) transform this graph to expose parallelism, and (3) search for mappings of the transformed graph to the target processor. We next discuss these steps in detail.

4.3.1 Characterizing the Application

The Bamboo compiler combines the ASTGs for the individual classes into a combined state transition graph (CSTG) to characterize the entire application. It then annotates the nodes and edges in the CSTG with runtime profile information.

The Bamboo compiler uses profile data to obtain the analog of a developer's intuition about the behavior of applications. Bamboo supports generating single or many-core profiling versions of applications. Single-core profiling is used to bootstrap the application synthesis process. A profile includes cycle counts for task invocations, the task exit taken by each task invocation, and a count of the number of parameter objects a given task invocation allocated. The Bamboo compiler processes the profile data to compute statistics including the average execution time a task takes for a given exit, the probability that the task takes the exit, and the average number of new objects allocated when the task takes the exit.

Figure 3 from Section 2.4 presents an example CSTG. A solid rectangle in the graph represents a *core group* — all tasks in a core group will be mapped onto the same core. It also describes the possible state transitions of the objects on that core. Therefore, a CSTG represents a possible implementation of a computation on a many-core processor. The compiler next performs a series of transformations on the CSTG to optimize the implementation.

4.3.2 Preprocessing

A tree transformation phase preprocesses the CSTG to prepare it for subsequent parallelization phases. We note that core groups may have more than one incident new object edge. These edges represent disjoint sources of work for the core group and present an opportunity for parallelism — the compiler can replicate the core group for each source of work. This phase transforms the CSTG into a tree of strongly connected components (SCCs) by duplicating SCCs as necessary.

The algorithm begins by computing the SCCs in the CSTG. For the purpose of computing SCCs in the CSTG, the compiler conceptually inserts a task node on each task transition edge that serves as the source of all the new object edges associated with that task transition. It then duplicates SCCs which have more than one incident edge originating from different SCCs. This process continues until each core group (except the `StartupObject` class core group) has exactly one incident new-object edge.

4.3.3 Parallelizing the Implementation

The Bamboo language is implicitly parallel. This phase transforms the CSTG to make the parallelism inherent in the application explicit. It is structured as a set of rules, where each rule transforms the CSTG to address an opportunity to improve performance. The compiler implements the following transformation rules:

- **Data Locality Rule:** The default rule maximizes data locality by placing tasks on the same core unless other rules apply. This minimizes communications to coordinate the task invocation. Moreover, this optimization is likely to improve performance due to caching.
- **Data Parallelization Rule:** If a task in one core group creates objects of a class that is processed by a second core group, task invocations on these new objects can potentially be processed in parallel with task invocations in the first core group.

  Profile information for allocation sites contains the expected number m of objects a given task invocation will allocate. The compiler then replicates the destination core group to generate $m - 1$ new copies.
- **Rate Matching Rule:** Short cycles in a CSTG that produce new objects can overwhelm a consumer core group's ability to process these objects. We introduce a rule that replicates the consumer core group as necessary to match the object consumption rate with the creation rate. We apply this rule only if the source core group is in a different SCC than the destination core group. Given the expected number m of allocated objects on the allocation site from the profile, the peak new object creation rate is $\frac{m}{t_{\text{cycle}}}$ and the object consumption rate for n copies of the consumer core group is $\frac{n}{t_{\text{process}}}$.

  For a task A that allocates new objects, let t_{recycle} be the time of the shortest path from the destination of A to the source of A. $t_{\text{cycle}} = t_A + t_{\text{recycle}}$ is the shortest time to complete the cycle. t_{process} is the estimated object processing time for the consumer core group containing the new object created by A. Matching these rates we get $n = \lceil \frac{m t_{\text{process}}}{t_{\text{cycle}}} \rceil$. The compiler compares n to m. If m is greater, it applies the data parallelization rule. If n is greater, it applies the cycle rate matching rule to generate $n - 1$ new copies of the destination core group.

4.3.4 Mapping to the Processor

We next discuss how the compiler maps an optimized CSTG to layouts for a physical processor. The mapping process uses a backtracking-based search algorithm to generate non-isomorphic mappings of the SCCs of core groups to the cores. We have extended the standard enumeration algorithm to randomly skip subsets of the search space. Thus, the extended algorithm generates a random set of non-isomorphic mappings.

For each mapping, the compiler generates a candidate layout. Figure 4 from Section 2.4 presents an example layout. The layout specifies for each core: (1) which tasks are on the core and (2) a table that lists for each destination abstract object state that may be generated by tasks on the core where the core should send the object. If there are multiple destinations for the same abstract object state, the runtime distributes the objects in a round-robin fashion.

An issue arises if the generated layout includes more than one instantiation of a task that operates on multiple parameter objects. Such a task can fail to trigger even if objects are available to serve as all of its parameter objects because the parameter objects could be enqueued in different instantiations of the task. If the task declaration requires that all parameter objects are tagged with the same tag, Bamboo hashes the tag instances bound to an object to determine which core to send the object to. Otherwise, it only generates one instantiation of the task and statically chooses one core group to process this instantiation. The host core can be specified by the developer or randomly chosen by the compiler.

4.4 Performance Estimation

The candidate implementation generation stage creates many candidate layouts for an application. A high-level discrete-event simulation estimates the relative performance of these candidate layouts using profile information. The simulation strategy was designed to support future extensions that would allow the synthesis process to use detailed specifications of an individual core's capabilities and a processor's on-chip network to optimize the executable. Note that the simulator does not actually execute the application — it instead uses profile data to estimate for a given layout how long the application will likely take to execute. We evaluate in Section 5.4, whether the final layout generalizes well to other inputs.

The simulation begins by injecting a startup object into the heap. The simulator then checks if there is a task that can be executed on any core. When a task can be executed, the simulator uses a Markov model built from the profile to estimate: (1) the destination state of the task, (2) the time taken to execute the task, and (3) a count of each type of new parameter object that the task allocates. The simulator maintains a count for each possible destination state of a task, which it increments when the simulated task takes the given transition. For each task invocation, the simulator chooses the destination state that minimizes the difference between these counts and the counts predicted by the task's recorded statistics. The simulator can accept developer hints that specify for a given task whether the counts are maintained on a per object basis or per task basis. It estimates the task execution time using an average of the execution times for a given exit point of the task.

The simulator then skips ahead to the finish time of the currently executing task that will finish first. The simulator state is updated to reflect the completed task. If new objects were created, the simulator generates new object events for the destination cores.

4.5 Optimizing with Directed-Simulated Annealing

For real world applications, the synthesis stage can potentially generate several million or more non-isomorphic candidate layouts. An exhaustive search of these layouts is infeasible for most applications. One possible solution is to randomly generate candidate implementations to evaluate. Section 5.3 presents experimental results that show that implementations with good performance are rare for our benchmark applications and therefore randomly generating implementations is unlikely to yield well-optimized implementations.

Instead, Bamboo combines random generation of several initial candidate layouts with a directed-simulated annealing based optimization algorithm. The directed-simulated annealing algorithm improves the candidate layouts to generate an optimized layout. The design mirrors actions taken by real developers — a developer executes an application, analyzes the execution to identify possible opportunities for optimization, implements the optimizations, and then repeats the process until she obtains the desired performance.

Our directed-simulated annealing algorithm operates in an iterative fashion — in each iteration it identifies performance problems in the candidate layouts and then generates several new layouts designed to correct those performance problems. Each iteration begins by running the simulation on the set of candidate layouts. The algorithm then prunes the set of candidate layouts using a probabilistic strategy: it keeps the best layouts with a high probability and poor layouts with a small probability. A critical path analysis for each execution identifies possible opportunities to further improve the candidate layout (see Section 4.5.1). The compiler uses the results of this analysis to generate a new set of candidate layouts that have been modified to attempt to remove the bottleneck

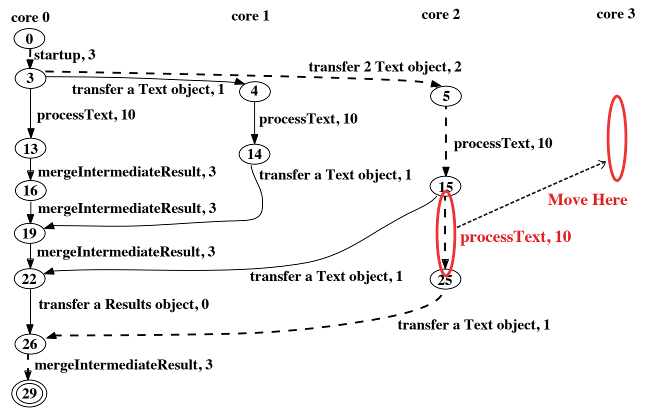

core 0
core 1
core 2
core 3
0
startup, 3
3
transfer 2 Text object, 2
transfer a Text object, 1
4
5
processText, 10
processText, 10
processText, 10
13
14
mergeIntermediateResult, 3
transfer a Text object, 1
15
Move Here
16
mergeIntermediateResult, 3
processText, 10
19
mergeIntermediateResult, 3
22
transfer a Text object, 1
25
transfer a Results object, 0
transfer a Text object, 1
26
mergeIntermediateResult, 3
29

Figure 6. Execution Trace for the Keyword Counting Example

in the previous candidate (see Section 4.5.2). This iterative process is repeated until the current best layout has the same or better performance than the candidate layouts from the previous iteration. Because the optimization process may have simply reached a local maxima, the algorithm probabilistically decides whether to continue searching or not, with a high probability to continue.

The primary difference between our directed-simulated annealing algorithm and the standard simulated annealing algorithm is that we use the results of the critical path analysis to direct the generation of future optimized candidate layouts. Section 5.3 presents our evaluation of the directed-simulated annealing algorithm.

4.5.1 Critical Path Analysis

We next describe the critical path analysis used to direct the generation of new candidate layouts. The critical path analysis processes execution traces generated from the simulated execution of the layouts. Figure 6 presents an execution trace for the keyword counting example. Nodes represent events in the simulated execution on the cores. Node labels describe when the events happened. Edges represent either task invocations or data transfers between cores. There are edges between (1) the nodes corresponding to the start and end of a given task invocation, (2) the end node of one task and the start node of the next task on the same core if the invocation of the second task had to wait for the completion of the first task, and (3) the end node of one task and the start node of a second task if the invocation of the second task had to wait for data from the first task. Weights are associated with each edge to indicate how long it takes to execute the task instance or transfer the data. Edge labels give the name of the task or the transferred data and the weight.

The dashed edges in Figure 6 indicate the critical path of this graph. It is the path with the largest weight from the start of the execution to the end of the execution. Note that this critical path accounts for both resource and scheduling limitations.

4.5.2 Optimizing Implementations

For each task instance on the critical path, the optimization algorithm computes the time when its data dependencies are resolved, which is the earliest point when all the parameter objects of the task instance are ready. The algorithm sorts task instances by the data dependence resolution time. Task invocations whose data dependencies are resolved at the same time compete with each other for computational resources. The algorithm groups such task instances together. Next, it randomly selects a group to attempt to optimize.

A difference between the time at which a task instance's data dependences are resolved and when the task is executed implies that the task instance was delayed because of a resource conflict. If there are spare cores during the interval between a task instance's expected start time and its actual start time, the optimization algorithm attempts to shorten the critical path by generating a set of new layouts in which the task instance is migrated to a spare core.

When spare cores are not available, moving tasks to other cores can still be desirable. In this case, the optimization algorithm identifies a set of *key task instances* — tasks on the critical path that produce data that the next task on the critical path consumes. Carefully scheduling key task instances is likely to be more important than other tasks as there are tasks that depend on the data that key tasks produce. The algorithm identifies situations in which a non-key task instance on the critical path delays the invocation of a key task instance. It attempts to move the non-key task instance to other cores to eliminate the resource conflict.

The algorithm extends the core search algorithm described in Section 4.3.4 to generate the new candidate layouts which redeploy the chosen task invocations on selected cores. It then iteratively repeats the simulation, evaluation, and optimization process until several iterations fail to yield any improvements.

4.6 Comparison to Dynamic Scheduling

An alternative to our approach is to dynamically schedule tasks using a centralized scheduler. Our approach has several key advantages. The first advantage is that as the number of cores increases, a centralized scheduler will quickly become the performance bottleneck. Our approach generates implementations that distribute the work of scheduling tasks across all cores. The second advantage is that our approach can generate sophisticated implementations that account for future data dependencies. For example, we have observed that our approach generates implementations of the MonteCarlo benchmark that use pipelining to overlap the simulation and aggregation components of the computation. Moreover, it is straightforward to extend our basic approach to optimize for data locality, heterogeneous cores, and new network topologies by simply extending the simulation to model these factors.

4.7 Runtime System

In general Bamboo uses a similar runtime strategy as earlier work on Bristlecone [17]. The primary differences are that the scheduler is distributed across all cores, the finite state machines generated by the static analysis are used to optimize task dispatch, Bamboo does not support task rollback or recovery, and the static analysis is used to resolve many inter-core scheduling decisions.

Each processor core runs a lightweight Bamboo runtime that schedules tasks for that core. Each task on a core has a parameter set for each parameter — objects that may satisfy the task's parameter guard are placed in the corresponding parameter set.

For each combination of task and parameter object, the combined state transition graph shows the set of tasks that can be invoked next on that parameter object. The compiler generates customized code for each task that sends a message directly to the cores that execute the next tasks to add the object to the appropriate parameter sets. When a new object is added to a parameter set, the runtime enqueues new task invocations (assignments of parameter objects to parameters) that the new object makes possible. The runtime contains optimizations to efficiently task dispatch with tags constraints. Each tag instance contains backward references to all objects it is bound to. The runtime uses these references to efficiently prune task invocations that contain tag constraints.

The runtime selects task invocations to execute from the queue of task invocations. Before executing a task invocation, the runtime locks all of the parameter objects. If it is unable to acquire a lock, it releases all locks and tries a different task invocation.

5. Evaluation

We have developed a Bamboo implementation, which contains approximately 120,000 lines of Java and C code for the compiler and runtime system. The compiler generates C code that runs on the TILEPro64 many-core processor. The TILEPro64 processor contains 64 cores interconnected with an on-chip network. The source code for our compiler is available at `http://demsky.eecs.uci.edu/compiler.php`. We executed our benchmarks on a 700MHz TILEPro64 processor. We used 62 cores as 2 cores are dedicated to the PCI bus.

For each benchmark, we generated three versions: a single-core C version, a single-core Bamboo version, and a 62-core Bamboo version. We executed these three versions on the TILEPro64 processor and recorded how many clock cycles were taken for each execution. Figure 7 presents the results. The reported execution times are averaged over five executions.

Benchmark	Clock Cycles($10^8 cyc$)			Speedup to 1-Core Bamboo	Speedup to 1-Core C	Overhead of Bamboo
	1-Core C	1-Core Bamboo	62-Core Bamboo			
Tracking	405.2	406.4	15.5	26.2	26.1	0.3%
KMeans	1124.6	1243.8	32.0	38.9	35.1	10.6%
MonteCarlo	44.4	47.0	1.3	36.2	34.2	5.9%
FilterBank	554.6	554.9	14.8	37.5	37.5	0.1%
Fractal	162.5	172.6	2.8	61.6	58.0	6.2%
Series	1774.7	1885.7	30.8	61.2	57.6	6.3%

Figure 7. Speedup of the Benchmarks on 62 cores

5.1 Results

We report our results on six benchmarks:

- **Tracking** The tracking benchmark extracts motion information from a sequence of images. It was ported from the San Diego Vision benchmark suite [32].

  Figure 8 shows the task flow of the Bamboo version. The vertical lines divide the task flow into the three major computation phases: image processing, feature extraction, and feature tracking. Each node represents a task. Edges show how data flows between the tasks. Dashed boxes group related tasks together.

  The benchmark utilizes data parallelism: the image is divided into multiple pieces and each piece is wrapped with a task parameter object. The computations in the dashed boxes work on pieces and then later aggregate these results for these pieces.

  The speedup of the 62-core Bamboo version is 26.2× relative to the single-core Bamboo version and is 26.1× relative to the single-core C version.

- **KMeans:** The KMeans benchmark groups objects in an N-dimensional space into K clusters. The algorithm is used to partition data items into related subsets. We ported it from the STAMP benchmark suite [9]. Our implement differs from the original version in that it does not use transactions to update the shared data structures. Instead, one core runs a task to update this data structure, and the other cores send partial results to that core. The speedup of the 62-core Bamboo version is 38.9× relative to the single-core Bamboo version and is 35.1× relative to the single-core C version.

- **MonteCarlo:** The MonteCarlo simulation benchmark was ported from the Java Grande benchmark suite [29]. It implements a Monte Carlo simulation. The speedup of the 62-core Bamboo version is 36.2× relative to the single-core Bamboo version and is 34.2× relative to the single-core C version.

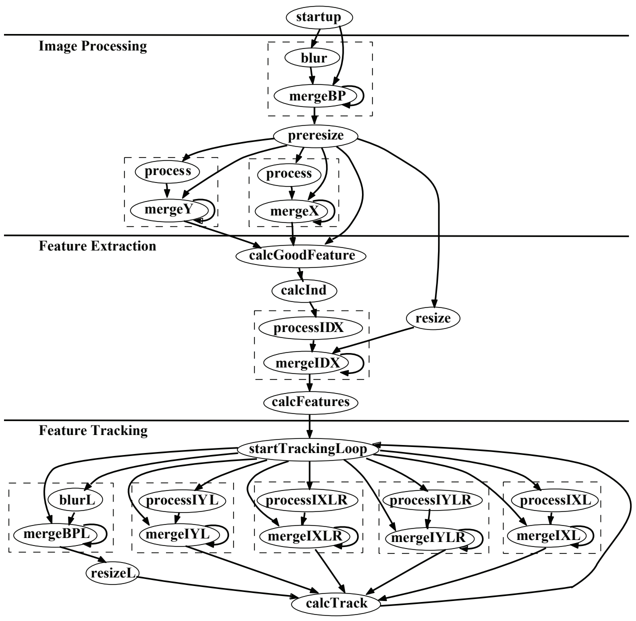


Figure 8. Task Flow of the Tracking Benchmark

  We were surprised to find that for larger workloads Bamboo generated a sophisticated heterogeneous implementation that used pipelining to improve performance by overlapping simulation and aggregation. We further discuss this in Section 5.4.

- **FilterBank:** FilterBank is a multi-channel filter bank for multi-rate signal processing. We ported this benchmark from the StreamIt benchmark suite [20]. It performs a down-sample followed by an up-sample on each channel and then combines the results for all channels. The speedup of the 62-core Bamboo version is 37.5× relative to the single-core Bamboo version and is 37.5× relative to the single-core C version.

- **Fractal:** Fractal computes a Mandelbrot set. We observed a 61.6× speedup of the 62-core Bamboo version relative to the single-core Bamboo version and a 58.0× speedup relative to the single-core C version.

- **Series:** Series computes Fourier coefficients. We ported it from the Java Grande benchmark suite [29]. The speedup of the 62-core Bamboo version is 61.2× relative to the single-core Bamboo version and 57.6× relative to the single-core C version.

The directed-simulated annealing algorithm took 1.3 minutes to optimize the Tracking benchmark, 10 seconds for the KMeans benchmark, and less than 0.2 seconds for the other benchmarks. The Bamboo compiler was executed on a quad-core 2.00 GHz Intel Xeon running 64-bit Linux version 2.6.18.

5.2 Accuracy of Scheduling Simulator

The accuracy of the high-level scheduling simulator is important as the final implementation is selected based on the scheduling simulation results. To evaluate the accuracy of the scheduling simulator, we compared the estimated execution time for the 62-core Bamboo implementation strategy chosen by the scheduling simulator with the real execution time of the corresponding 62-core binary for each of our benchmarks.

Benchmark	1-Core Bamboo Version			62-Core Bamboo version		
	Clock Cycles($10^8 cyc$)			Clock Cycles($10^8 cyc$)		
	Estimation	Real	Error	Estimation	Real	Error
Tracking	405.9	406.4	-0.1%	14.9	15.5	-3.9%
KMeans	1265.1	1243.8	1.7%	31.9	32.0	-0.3%
MonteCarlo	47.1	47.0	0.2%	1.2	1.3	-7.7%
FilterBank	554.8	554.9	-0.02%	14.1	14.8	-4.7%
Fractal	170.7	172.6	-1.1%	2.8	2.8	0.0%
Series	1856.7	1885.7	-1.5%	29.9	30.8	-2.9%

Figure 9. Accuracy of Scheduling Simulator

Figure 9 presents the results. The simulation's predictions are close to the real execution time. For MonteCarlo, the estimation for the 62-core Bamboo version is 7.7% less than the real execution time. A closer examination of the profiling data shows that when executing on 62 cores, the execution of individual tasks slowed down. The 4.7% difference for FilterBank has a similar cause.

5.3 Efficiency of Directed-Simulated Annealing

We next discuss our evaluation of the directed-simulated annealing used to efficiently optimize the many-core implementations. In this experiment, we exhaustively generated all candidate implementations. We did the evaluation on 16 cores instead of 62 cores because an exhaustive search of all candidate implementations for 62 cores is prohibitively expensive. For each benchmark, we selected an input and collected the corresponding profiling data. For most benchmarks, we first generated all possible candidate implementations and used the scheduling simulator to evaluate them. We did not perform this experiment for the Tracking benchmark as an exhaustive search for even 16 cores is prohibitively expensive. The empty bars in Figure 10 present result of this experiment which show the probability distributions for the candidate implementations. The x-axis is the estimated execution time of the candidate implementation. The y-axis is the relative percentage of the particular estimated execution time. Candidate implementations with the smallest estimated execution times are the best. The graphs show that for most benchmarks there is a very small chance of randomly generating the fastest implementation.

We next show that directed-simulated annealing greatly increases the probability of synthesizing the fastest implementation. For each benchmark, we randomly chose 1,000 candidate implementations as starting points for the directed-simulated annealing algorithm. For each starting point, we executed the directed-simulated annealing and recorded the execution time of the best implementation generated by the directed-simulated annealing algorithm. The solid bars in Figure 10 present the probability distributions for this experiment. We found that with directed-simulated annealing, the probability of generating the best candidate implementation from a random starting point is larger than 98% for all benchmarks. For KMeans, the probability reaches 100%.

5.4 Generality of Synthesized Implementation

We used profiling data to generate an implementation that is optimized for the target many-core processor. We expect that if the profile data exposes sufficient parallelism in the application, the optimized implementation will work well for inputs larger than the input for the profiled execution.

We next discuss our evaluation of how well our optimized implementation generalize to other inputs. For each benchmark, we define the original input as $\text{Input}_{\text{original}}$ and defined a second input $\text{Input}_{\text{double}}$, which contains a workload that is twice as large. We then collected profiling data $\text{Profile}_{\text{double}}$ for $\text{Input}_{\text{double}}$ and generated a new 62-core Bamboo version using the $\text{Profile}_{\text{double}}$. We executed the new version as well as the single-core Bamboo version and the 62-core Bamboo version generated with $\text{Profile}_{\text{original}}$ on the new $\text{Input}_{\text{double}}$. The results are presented in Figure 11.

Benchmark	$\text{Profile}_{\text{original}}$, $\text{Input}_{\text{double}}$			$\text{Profile}_{\text{double}}$, $\text{Input}_{\text{double}}$	
	Clock Cycles ($10^8 cyc$)		Speedup	Clock Cycles ($10^8 cyc$)	Speedup
	1-Core	62-Core		62-Core	
Tracking	1594.0	44.8	35.6	44.7	35.7
KMeans	5147.9	125.8	40.9	125.5	41.0
MonteCarlo	94.1	2.6	36.2	1.8	52.3
FilterBank	1109.6	19.9	55.8	19.9	55.8
Fractal	289.8	5.8	50.0	5.1	56.8
Series	3785.4	61.3	61.8	63.6	59.5

Figure 11. Generality of Synthesized Implementations

For most benchmarks, the speedup of both 62-core Bamboo versions are similar indicating that the synthesized binaries generalize to different inputs. For MonteCarlo, the 62-core version generated using $\text{Profile}_{\text{double}}$ performs much better on $\text{Input}_{\text{double}}$ than the 62-core version generated using $\text{Profile}_{\text{original}}$. After examining the two versions, we were surprised to discover that our search-based synthesis algorithm generated a heterogeneous implementation that utilized pipelining to overlap the aggregation and simulation computations. The smaller input size does not contain enough work to benefit from the pipelining strategy, and therefore did not yield a pipelined implementation. We examined Fractal and Series and discovered that the differences between the speedups are due to differences in the workloads on the cores, which occurs because object are distributed in different orders for the two versions.

5.5 Overhead of Bamboo

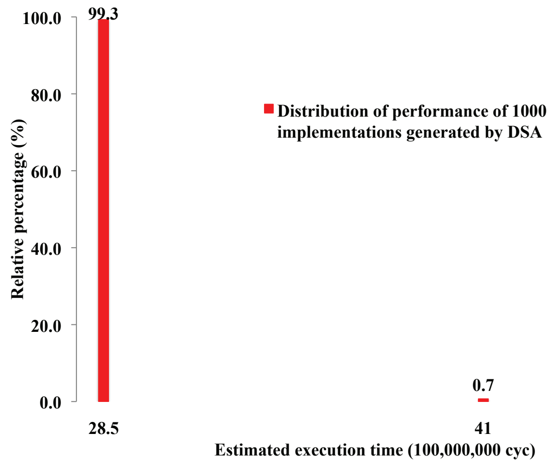
To characterize the overhead of the Bamboo language and runtime, we compared the performance of the single-core C version and the single-core Bamboo version for each benchmark. Results are listed in Figure 7. Bamboo optionally supports array bounds checks for non-performance critical applications. We turned off the array bounds check option for these benchmarks so as to be comparable with the C versions. For Tracking, KMeans, MonteCarlo, FilterBank, Fractal, and Series, we found that the overheads of Bamboo are 0.3%, 10.6%, 5.9%, 0.1%, 6.2%, and 6.3%, respectively.

5.6 Discussion

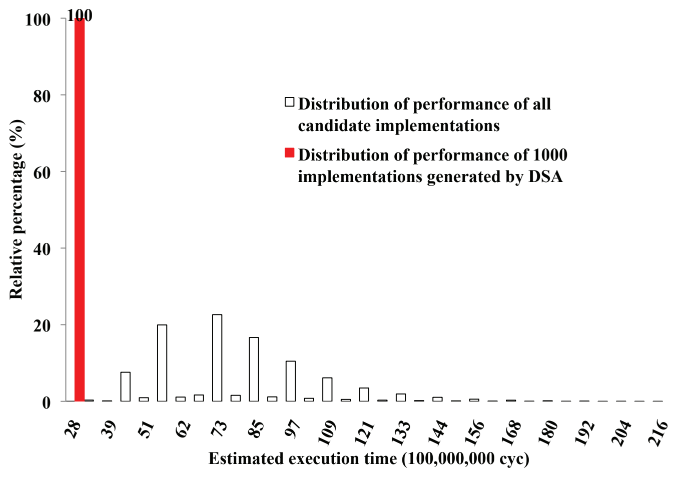
We evaluated several aspects of the Bamboo implementation synthesis tool. We found that the parallel implementations that the Bamboo compiler generated not only achieved significant speedups when compared to the single-core Bamboo/C implementations, but they also generalized to other sized inputs. We found that the scheduling simulator generated accurate and useful estimations for the evaluation of candidate implementation strategies, and the directed-simulated annealing algorithm greatly helps to efficiently generate optimized implementations.

Moreover, we found that the implementation synthesis tool generated a sophisticated implementation for the Monte Carlo simulation. We were surprised by the synthesized implementation as we had not realized that the benchmark's performance could be improved by overlapping the simulation and aggregation components.

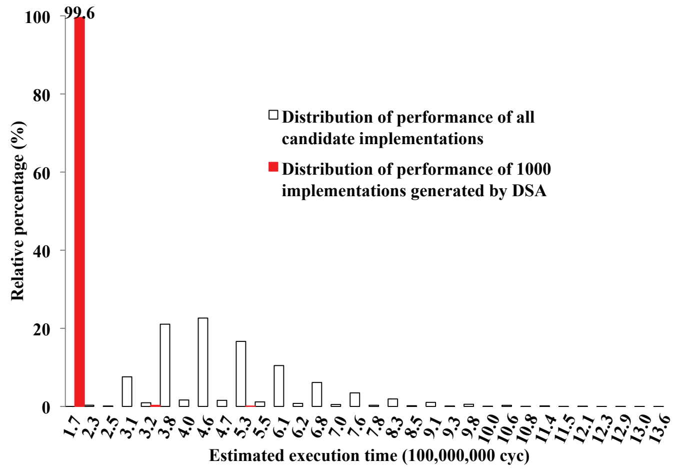
We found that the runtime overheads of Bamboo relative to C were relatively small. We also found porting the applications to Bamboo to be straightforward and freed us from many of the low-level concerns of writing parallel code in C. The porting process simply involved structuring the program as a set of tasks and writing a few short task declarations to describe the dependencies between the tasks.



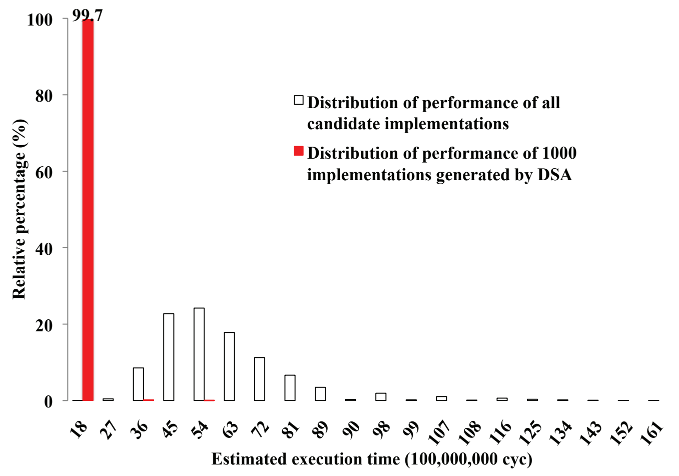
(a) Tracking



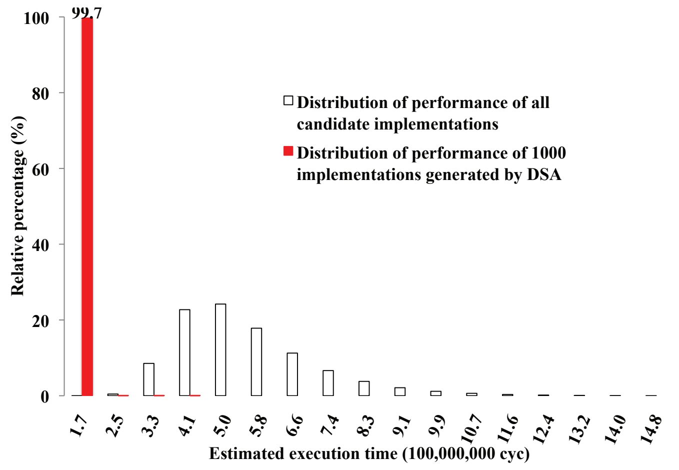
(b) KMeans



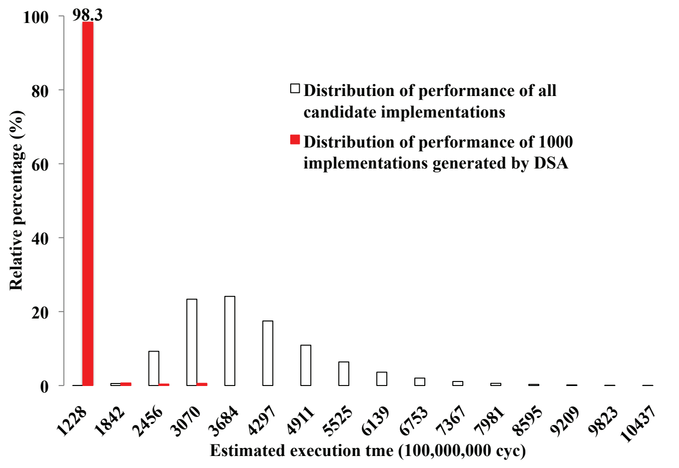
(c) MonteCarlo



(d) FilterBank



(e) Fractal



(f) Series

Figure 10. Efficiency of Directed-Simulated Annealing Algorithm

6. Related Work

Researchers have developed parallelizing compilers, domain specific languages, explicitly parallel languages [5, 11, 12], work-stealing based multi-threaded systems [18], nested data-parallel languages [8], array-based programming languages [10], and other external tools for developing efficient parallel software. We survey related work in languages and automatic parallelization.

A key component of Bamboo is decoupling unrelated conceptual operations and tracking data dependencies between these operations. Dataflow computations also keep track of data dependencies between operations so that the operations can be parallelized [24]. Bamboo borrows ideas from dataflow and integrates them within the context of a standard imperative language to ease adoption by developers. Bamboo relaxes key restrictions in the dataflow model to permit flexible mutation of data structures and construction of structurally complex data structures. Furthermore, Bamboo supports applications that non-deterministically access data.

Course-grained or macro-dataflow languages [14, 22] compose several sequential operations together to construct larger granularity code segments for dataflow execution. Relative to these languages, Bamboo provides a safe mechanism to support arbitrary mutation of structurally complex data structures. Bamboo also statically generates distributed runtime schedulers that are optimized for a program's typical runtime behavior.

Tuple-space languages, such as Linda [19], decouple computations to enable parallelization. The threads of execution communicate through primitives that manipulate a global tuple space. Because these threads can contain state, the compiler cannot automatically create multiple instantiations to utilize additional cores.

The orchestration language Orc [13] specifies how work flows between tasks. Note that if an operation fails, any work (and any corresponding data) flowing through the task may be lost. Another language, Oz, is a concurrent, functional language that organizes computations as a set of tasks [30]. Tasks are created and destroyed by the program. Task reducibility is monotonic — once a task is reducible it is always reducible. Ada [1] also has tasks. However, Ada tasks have state and therefore are not straightforward to parallelize.

Bamboo's computational model is similar to actors. Actors communicate through messages [3, 21]. The parallelism in an actor-based program is limited by the number of actors the developer created. We note that because actors contain state, an individual actor is not straightforward to parallelize.

Plaid extends object-oriented languages with support for typestate-oriented programming [4]. While the formulations and goals of Plaid differ from Bamboo, both projects leverage the power of making high-level state constraints visible to the compiler infrastructure. The two approaches differ in how they handle the problem of tracking typestate in the presence of aliasing: Plaid uses a type system to ensure reference uniqueness while Bamboo uses dataflow-like dispatch model.

Previous languages have use the concept of object states or views to express correctness constraints on concurrent accesses to objects [15, 16]. Constraints associated with object states can provide a natural expression of read-write locks. Bamboo's use of object states differs from this work in that Bamboo uses abstract object states to coordinate task invocation.

Jade provides annotations that developers use to specify how to decompose methods in a serial program into a set of tasks [28]. These annotations describe the data that a Jade task reads from or writes to along with commutativity properties. Jade uses these specifications to parallelize applications at runtime. The approaches are complementary — Bamboo can be used to extract less structured parallelism while Jade can be used to extract more structured parallelism from the imperative code inside of Bamboo tasks.

Streaming languages [20, 25] are designed to support computations that can be structured as streams. While Bamboo shares similar constructs with stream-based languages, Bamboo's task dispatch is considerably more expressive and eliminates key weaknesses of stream languages. For example, in stream languages it is difficult to express computations in which several different parts of the computation access a shared data structure in an irregular pattern. Bamboo's task dispatch supports irregular dispatch patterns on shared objects — the developer simply creates an instance of the shared object type and then uses the task specifications to specify the shared object. Bamboo also adds supports for sharing structurally complex data structures and mutating shared objects.

Spiral uses search to optimize DSP algorithms [27]. While both approaches use search, Bamboo targets general computation rather than Spiral's more specific focus on linear DSP algorithms.

Bamboo borrows constructs from the Bristlecone language for creating robust software systems [17]. Our previous work only supports single-threaded execution and contains language constructs specific to automated recovery. Bamboo shares language ideas with Bristlecone, but extends these ideas to support parallel execution.

CellSs dynamically schedules function invocation when a function's operands are available [7]. Bamboo differs in that it supports linked data structures, which are not supported by CellSs. Bamboo also differs in that we use static analysis to eliminate scheduling overheads and to parallelize the runtime scheduler. We expect that our distributed schedulers would scale to much larger processors than CellSs's centralized runtime scheduler.

Many of the recent efforts on software synthesis for parallel machines have focused on fully automatic approaches to parallelization (e.g., [6]). When this approach is effective, it is ideal because it maximizes programmer productivity. Bamboo is largely complementary to this work. It is conceptually straightforward to leverage parallelizing compilers to extract fine-grained parallelism by automatically parallelizing individual Bamboo tasks.

7. Conclusion

We have successfully implemented several parallel applications in Bamboo. Bamboo applications consist of a set of interacting tasks with each task implementing one of the conceptual operations in the application. The developer specifies how these tasks interact using task declarations. Bamboo extracts data dependence information from the declarations and combines this information with profile data to automatically synthesize parallel implementations that are optimized for a target many-core processor. Bamboo generated implementations of our benchmark applications that scaled successfully to 62 cores. The implementations generalized to other sized inputs. Moreover, Bamboo generated sophisticated implementation that used pipelining to overlap the computation and data aggregation phases of the benchmark applications.

Our current implement performs optimization at compile time. However, the basic technique is more generally applicable. It is straightforward to modify the basic approach to support executables that periodically re-optimize themselves for the workloads they encounter in the field or for new processor layouts. The basic idea is to separate layout information from code in the application executable. An executable would periodically profile itself and report the results to a system library that implements our optimization strategy. The library would then rerun the optimizations, generate a new layout, and update the executable's layout information.

Acknowledgments

This research was supported by the National Science Foundation under grants CCF-0846195 and CCF-0725350. We would like to thank the anonymous reviewers for their helpful comments.

References

[1] Ada Reference Manual. `http://www.adaic.org/standards/05rm/html/RM-TTL.html`, 2005.

[2] Tilera. `http://www.tilera.com/`.

[3] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[4] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. State-oriented programming. In *Proceedings of Onward!*, 2009.

[5] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Messen, S. Ryu, G. L. Steele, and S. Tobin-Hochstadt. *The Fortress Language Specification*. Sun Microsystems, Inc., September 2006.

[6] S. P. Amarasinghe, J.-A. M. Anderson, M. S. Lam, and A. W. Lim. An overview of a compiler for scalable parallel machines. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, 1994.

[7] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: A programming model for the Cell BE architecture. In *Proceedings of the ACM/IEEE SC 2006 Conference on Supercomputing*, 2006.

[8] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, Apr. 1994.

[9] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, September 2008.

[10] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby. The case for high level parallel programming in ZPL. *IEEE Computational Science and Engineering*, pages 76–86, July–September 1998.

[11] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 2007.

[12] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2005.

[13] W. R. Cook, S. Patwardhan, and J. Misra. Workflow patterns in Orc. In *Proceedings of the 2006 International Conference on Coordination Models and Languages*, 2006.

[14] K. Dai. Code parallelization for the LGDG large-grain dataflow computation. In *CONPAR 90/VAPP IV: Proceedings of the Joint International Conference on Vector and Parallel Processing*, pages 243–252, London, UK, 1990. Springer-Verlag.

[15] F. Damiani, E. Giachino, P. Giannini, N. Cameron, and S. Drossopoulou. A state abstraction for coordination in Java-like languages. In *Electronic Proceedings of the 2006 Workshop on Formal Techniques for Java-like Programs*, 2006.

[16] B. Demsky and P. Lam. Views: Object-inspired concurrency control. In *Proceedings of the 2010 International Conference on Software Engineering*, 2010.

[17] B. Demsky and S. Sundaramurthy. Bristlecone: Language support for robust software applications. *IEEE Transactions on Software Engineering*.

[18] M. Frigo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. In *International Conference on Programming Language Design and Implementation*, 1998.

[19] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[20] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffman, D. Z. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October, 2002.

[21] C. Hewitt and H. G. Baker. Actors and continuous functionals. Technical report, Massachusetts Institute of Technology, 1978.

[22] C. Huang and L. V. Kale. Charisma: orchestrating migratable parallel objects. In *Proceedings of the 2007 ACM International Symposium on High Performance Distributed Computing*, pages 75–84, 2007.

[23] J. C. Jenista and B. Demsky. Disjointness analysis for Java-like languages. Technical Report UCI-ISR-09-1, 2009.

[24] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1), 2004.

[25] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2008.

[26] H. J. Larson and B. O. Shubert. *Probabilistic Models in Engineering Sciences*. Wiley, 1979.

[27] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.

[28] M. C. Rinard. *The Design, Implementation and Evaluation of Jade, a Portable, Implicitly Parallel Programming Language*. PhD thesis, Stanford University, September 1994.

[29] L. A. Smith, J. M. Bull, and J. Obdrzalek. A parallel Java Grande benchmark suite. In *Proceedings of SC2001*, 2001.

[30] G. Smolka. The Oz programming model. In *Proceedings of the European Workshop on Logics in Artificial Intelligence*, page 251, London, UK, 1996. Springer-Verlag.

[31] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, January 1986.

[32] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The San Diego Vision Benchmark Suite. In *Proceedings of the IEEE International Symposium on Workload Characterization*, October 2009.