Automatically Tuning Task-Based Programs for Multi-core Processors

Jin Zhou and Brian Demsky
University of California, Irvine

Abstract. We present a new technique to automatically optimize parallel software for multi-core processors. We have implemented the technique for Bamboo, a task-based extension to Java. Optimizing applications for multi-core processors requires balancing the competing concerns of parallelism and communication costs. Bamboo uses high-level simulation to explore how to best trade off these competing concerns for an application. The compiler begins by generating several initial candidate implementations. The compiler then uses high-level simulation with profile statistics to evaluate these candidate implementations. It uses an as-built critical path analysis to automatically identify opportunities to improve the candidate implementation and then uses directed simulated annealing to evaluate possible optimizations.

We have implemented a Bamboo compiler. We have evaluated our implementation on four benchmarks: Series, a Fourier series computation; MonteCarlo, a Monte Carlo simulation; FilterBank, a multi-channel filter bank; and Fractal, a Mandelbrot set computation. We found the automatically optimized implementations had speedups of up to a factor of 14.7 when executed on 16 cores.

1 Introduction

Historically, developers have coded for simple, relatively static architectures. We expect this will change — in the future as fabrication technologies advance, the number of cores in each successive generation of microprocessors will increase. We have seen the beginning of this trend — every few months Intel releases a new generation of processors with more cores. Moreover, the types of cores, communication primitives, and memory systems in future processors will continue to change as processors evolve.

Tuning parallel software for multi-core processors using the current approaches presents many challenges. The developer must manually partition the computation across the processors. This requires the developer to reason about the critical paths in the computation, explore tradeoffs between parallelization and communication, and organize the computation to make optimal use of processor cores. Moreover, the software developer will have to repeat this entire procedure as processors evolve.

This paper presents a new approach for automatically tuning software for multi-core processors. The approach first generates a profiling version of the application that the developer uses to collect profile statistics. An implementation generation algorithm uses the profile statistics and a processor specification to explore opportunities for parallelizing the application and to synthesize a set of candidate implementations that serve as starting points for further optimization. During the optimization stage, it uses a combination of high-level simulation and directed-simulated annealing to automatically tune the candidate implementations to generate an executable that is optimized for the specific multi-core processor. As processors evolve, the developer simply reruns the automatic synthesis tool to generate a new binary that is optimized for the new processor.

This paper makes the following contributions:

- **Automatic Implementation Generation:** It introduces an algorithm that automatically generates many candidate multi-core implementations of the application.

- **High-level Simulation:** It introduces a simulation-based evaluation algorithm that the Bamboo compiler uses to help select optimized multi-core implementations.
- **Directed-Simulated Annealing:** It uses an as-built critical path analysis to direct a simulated annealing-based algorithm to optimize the candidate implementations. The combination of as-built critical path analysis and simulated annealing serves to quickly iterate through the process of identifying performance bottlenecks and exploring optimizations that would typically take considerable developer effort.
- **Experience:** It presents our experience using Bamboo on several benchmarks. We found that it was straightforward to develop Bamboo implementations of our benchmarks and that these implementations made effective use of multiple cores. Moreover, our algorithm generated a sophisticated implementation of the MonteCarlo benchmark that used pipelining to overlap the simulation and aggregation tasks.

2 Example

We next present a keyword counting example that illustrates the basic approach.

2.1 Task Extensions

Bamboo extends Java-like objects with *flags* to track the conceptual state of objects. Flags are declared in class declarations. Figure 1 presents part of the `Text` class declaration for the example. The keyword counter uses instances of the `Text` class to count the occurrences of words in a section of the input text. The `Text` class contains two flags: the `process` flag, to indicate that the `Text` object is ready to be processed, and the `submit` flag, to indicate that `Text` object has been processed.

Bamboo applications are structured as a collection of interacting *tasks*. Bamboo uses an object's conceptual state as indicated by its flags to determine which tasks to invoke on it. When the heap contains objects with the specified flag settings to serve as the task's parameters, the runtime invokes that task. When a task exits, it can change the values of the flags of its parameter objects. The key difference between tasks and methods is that the runtime controls task invocation. The runtime invokes a task when it can find objects in the heap that can serve as the task's parameters. Note that tasks can call methods through the standard mechanism.

Each task declaration consists of the keyword `task`, the task's name, the task's parameters, and the body of the task. The guards contain a set of predicates on the flags of the parameter objects. Only objects whose flags satisfy the task's guards can serve as a parameter object to the task. Tasks can in turn modify the state of an object's flags when (1) the object is allocated or (2) at the completion of the task's execution. Bamboo contains a *tag* construct that a developer can use to ensure that a task is dispatched on the correct group of objects.

Figure 2 presents the task declarations for the example. We indicate the omission of standard Java code inside the tasks with ellipses. 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.

2.2 Execution

This example performs the following operations (although not necessarily in this order):

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

Fig. 1. Text Class Declaration

```
task startup(StartupObject s in initialstate) {
  ...
  Partitioner p = new Partitioner(s.args[0]) {partition:=true};
  taskexit(s: initialstate:=false);
}
task partition(Partitioner p in partition) {
  ...
  while(p.morePartitions()) {
    String section = p.nextpartition();
    Text tp=new Text(section) {process:=true};
  }
  Results rp=new Results(nsection) {finished:=false};
  taskexit(p: partition:=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);
}
```

Fig. 2. Flag Specifications for Tasks

1. **Startup:** When a Bamboo program is executed, the runtime creates a `StartupObject` object and sets its `initialstate` flag to true. This causes the runtime to invoke the `startup` task. This task creates a `Partitioner` object to partition the incoming stream for further processing. It then resets the `initialstate` flag to false to prevent repeated invocation of the `startup` task.
2. **Partitioning the Text Stream:** The runtime invokes the `partition` task on a `Partitioner` object if its `partition` flag is true. This task partitions the text stream into sections, creates a new `Text` object for each section, and sets their `process` flags to true indicating that they are ready for processing. It then creates a `Results` object to merge the intermediate results.
3. **Processing a Text Section:** When the runtime identifies a `Text` object whose `process` flag is true, it invokes the `processText` task on the object to process the text section and stores the intermediate result in the object. Upon exiting, the `submit` flag is set to true to indicate that the intermediate result can be merged into the final result. Then the `process` flag is set to false to prevent the runtime from repeatedly invoking the `processText` task on the same object.
4. **Merging Results:** The `mergeIntermediateResult` task merges the intermediate results from the `Text` objects into the `Results` object. It sets the `Text`'s `submit` flag to false to prevent merging it again. If it has merged all of the intermediate results, it sets the `Results` object's `finished` flag to true.

2.3 Scheduling for Multi-core Processor

The Bamboo compiler uses an automatic implementation synthesis framework to generate and evaluate many possible candidate implementations of the application to synthesize an optimized multi-core implementation. This framework uses a *Combined Flag*

State Transition Graph (CFSTG) to reason about the possible behaviors of the application. This graph is automatically generated by a static analysis. Figure 3 presents the CFSTG for the example. The nodes in this graph model the abstract flag states for task parameter objects. Nodes with double ellipses indicate that newly allocated objects can be created with the abstracted state. The solid edges model the state transitions caused by the invocation of a task on an object. The creation of new objects is modeled with dashed edges — a new object edge points from the task that creates an object to the flag state node that abstracts the state of the new object. A rectangle in the CFSTG represents a *core group* that indicates which tasks in the CFSTG are mapped onto the same core — all of the computations inside a rectangle are mapped onto a single core. Thus a CFSTG represents a possible mapping of a computation onto a multi-core processor.

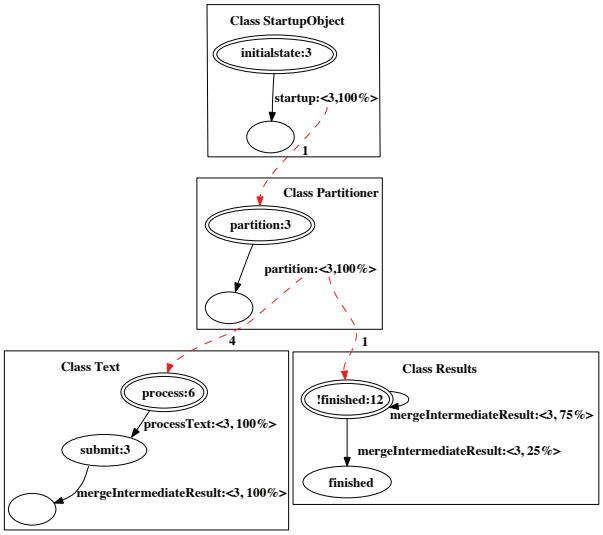

Fig. 3. CFSTG for the Keyword Counting Example

The compiler associates profile information with the nodes and edges in the CFSTG. The solid edges are labeled with the name of a task followed by a colon and a tuple that contains (1) the expected time the task takes to execute if it makes this transition and (2) the probability that the task will make 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 `partition` state is labeled `1` indicating that the task may generate 1 `Partition` object in the `partition` state. Each node contains a description of the abstract object state followed by a colon and an estimate of the lower bound of how long it will take to complete processing objects in this state.

The compiler transforms the CFSTG to optimize the application's implementation for the given processor. For example, it might begin by generating multiple instantiations of the tasks that process a `Text` object. The compiler uses these transformations to generate candidate implementations to serve as starting points for the directed simulated annealing optimization algorithm. The algorithm first runs a high-level simulation to evaluate these implementations using gathered statistics. Then it computes the as-built critical path of the simulated execution and uses the as-built critical path along

with dependence information to identify a potential set of tasks that can be migrated to different cores to reduce the length of the as-built critical path. It then generates a set of candidate implementations that implement these transformations and uses these candidate implementations as the starting point for iterative optimization procedure. When it reaches a point of diminishing returns, the compiler selects the best candidate implementation and generates the corresponding multi-core binary.

Figure 4 presents a candidate implementation 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.

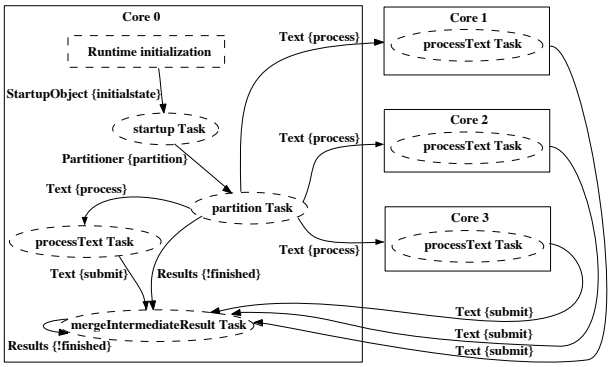

Fig. 4. Scheduling Plan on a 4-core Processor

3 Implementation Synthesis

We next describe the staged strategy used by the Bamboo compiler to automatically synthesize application implementations. The strategy contains the following stages:

- **Dependence Analysis:** The first stage statically analyzes the task specifications to characterize the application's data dependencies.
- **Disjointness Analysis:** The second stage statically analyzes the imperative code inside of Bamboo tasks and methods to determine whether the parts of the heap reachable from task parameter objects are disjoint[1]. The compiler uses this information to generate lock operations that collectively provide transactional semantics for tasks.
- **Candidate Implementation Generation:** The third stage synthesizes a random set of non-isomorphic candidate implementations. These candidate implementations serve as starting points for further optimization.
- **Simulation-based Evaluation:** The fourth stage uses profile information along with a processor specification to perform a high-level simulation of the generated implementation strategy. 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.
- **Directed Simulated Annealing:** Based on the evaluation results, the final stage uses directed simulated annealing to iteratively optimize the candidate implementations.

3.1 Dependence Analysis

The dependence analysis operates on the *flag state transition graphs* (FSTGs) [2]. A FSTG is associated with a flagged object type and abstracts the possible state transitions of instances of that type. A FSTG is composed of *flag state nodes* and edges between these nodes. A flag state node abstracts an object's flag state — the abstraction contains the states of all the object's flags. If an object in the computation can reach a given flag state, the FSTG for that object's class contains the corresponding flag state node. The FSTG contains a set of edges that abstract the actions of tasks on objects. If a task can be invoked on an object, there are edges labeled with the task (and the number of the parameter to which the object is bound) from the flag state node that correspond to the object's initial flag state to the flag state nodes that correspond to each possible flag state to which the task can transition the object.

3.2 Profiling

Bamboo includes support for generating single or multi-core profiling implementations of applications. Single-core support is used to bootstrap the application synthesis process. The profiling versions record an execution trace that includes cycle counts for task invocations, the task exit taken by each task invocation, and a count of the number of flagged objects the task invocation allocated.

3.3 Candidate Implementation Synthesis

The next stage in the compilation process generates several candidate implementations for further evaluation and optimization. It is structured as three steps: (1) generating the CFSTG, (2) transforming the CFSTG to expose parallelism, and (3) searching mappings from the transformed CFSTG to the target processor.

CFSTG The Bamboo compiler begins by combining the FSTGs for the individual classes into a CFSTG for the entire application. Figure 3 from Section 2 presents an example CFSTG. The nodes in this graph model the abstract flag states for the classes that serve as task parameters. Nodes with two ellipses indicate that newly allocated objects can be created with the abstracted state. The solid edges model the state transitions caused by the invocation of a task on an object. We model the creation of new objects with dashed edges — a new object edge points from the task that creates an object to the flag state node that abstracts the state of the newly created object.

The compiler associates profile information with the nodes and edges in the CFSTG. The solid edges each contain a label 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 the given transition and the probability that the task will make this transition. The dashed edges are labeled with a tuple that gives the expected number of newly created objects.

A rectangle in the CFSTG represents a *core group*. The rectangle indicates which tasks in the CFSTG are mapped onto the same core — all of the computations inside a rectangle are mapped onto a single core. Therefore a CFSTG represents a possible mapping of a computation onto a multi-core processor.

Tree Transform The analysis begins by transforming the CFSTG into a tree of *Strongly Connected Components* (SCCs). The algorithm computes the SCCs in the CFSTG. For the purpose of computing SCCs in the CFSTG, 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 the given task transition.

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 represent an opportunity for parallelism — the compiler can replicate the core group to create a new core group for each source of work. The compiler duplicates core groups with more than one incident new object edge until each core group (except the `StartupObject` class core group) has exactly one such incident new object edge.

CFSTG Transformations The compiler next transforms the CFSTG to optimize the implementation for the application's parallelism. These transformations are structured as a set of rules each of which makes a transformation to the CFSTG that is designed to address a common opportunity to improve performance.

- **Data Locality Rule:** The default rule maximizes data locality by placing tasks on the same core unless other rules apply. This minimizes the communications to coordinate the invocation of tasks. Moreover, this optimization is likely to improve performance on many processors due to caching.
- **Data Parallelization Rule:** If a task in one core group creates objects of class that is processed by a different core group, task invocations on the new objects can potentially be processed in parallel with task invocations in the current core group. The compiler applies this rule whenever the time to process all of the future tasks on the current core group is greater than the time to transfer the new object to another core. The compiler examines the profile information to determine the expected number of allocated objects and replicates the consumer core group to generate enough new copies so that the new objects can be processed by different cores in parallel.
- **Rate Matching Rule:** We observe that short cycles in a FSTG that produce new objects can overwhelm a consumer core group's ability to process the newly created objects. We therefore introduce a rule that replicates the consumer core group as necessary to process the objects created by the cycle. We apply this rule only if the source core group is in a different SCC than the consumer core group.
  The compiler computes the peak new object creation rate and the object processing rate from the profile information. It then matches these rates to determine how many copies of the consumer core group are sufficient to process the new objects in time. It generates the new copies accordingly.

Mapping to the Processor In the previous transformation stage, the CFSTG was transformed to create multiple instantiations of tasks to parallelize the computation. The next step is to map the final CFSTG to the cores in the actual processor. The mapping process may generate many candidate implementations for further evaluation and optimization.

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 basic enumeration algorithm to randomly skip subsets of the search space. The extended algorithm generates a random set of non-isomorphic mappings.

3.4 Simulation Stage

The candidate implementation synthesis algorithm generates multiple candidate mappings. To identify the best one, Bamboo uses a high-level simulator to evaluate each candidate mapping. The simulator uses profile information from previous executions to

simulate the execution of a candidate mapping. The simulation strategy was designed to support extensions that would allow the synthesis process to use detailed information about an individual core's capabilities and the on-chip network from a processor specification to optimize the binary. Note that the simulator does not actually execute the program — it instead uses profile data to estimate for a given mapping how long the application will likely take to execute.

The simulator is architected as a set of core simulators. A core simulator simulates the execution of Bamboo tasks instantiations on a core. For each task mapped to a core, the core simulator contains queues of simulated objects for each of the task's parameters. Connections are built between the core simulators to simulate data transfer.

Given a mapping to evaluate, the compiler generates the corresponding simulator. The simulated time is set to zero and a startup object is injected into the simulation. Then each core simulator checks if there is a task that can be executed. When a task can be executed, the simulators use a Markov model [3] built from the recorded statistics 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 flagged 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 the given taskexit of the task.

The simulator calls each core simulator to determine the start and finish times for the currently pending task on it. Then the task with the earliest finish time is picked and the host core simulator completes the task and updates the core's state. If new objects are created, the core simulator generates new object events on the destination cores.

3.5 Directed-Simulated Annealing

The synthesis stage can potentially generate a large number of candidate implementations. An exhaustive search of these candidate implementations is infeasible for many applications. Instead, Bamboo generates a few candidate implementations and then uses directed simulated annealing to iteratively improve these candidate implementations to generate a good implementation without exhaustively exploring all implementations.

Our directed simulated annealing algorithm operates in an iterative fashion — in each iteration it identifies performance problems in the candidate implementations and then generates several new implementations designed to correct those performance problems. Each iteration begins by running the simulation on the set of candidate implementations. The algorithm then prunes the set of candidate implementations using a probabilistic strategy: it keeps the best implementations with a high probability but even poor implementations have a (smaller) chance to survive until the next iteration. The algorithm next runs an as-built critical path analysis on each execution to identify possible opportunities to further improve the candidate implementation. It uses the results of this analysis to generate a new set of candidate implementations that have been modified to attempt to remove the bottleneck from the previous candidate. Typically this iterative process is repeated until: (1) the new search space is empty or (2)

the current best implementation has the same or better performance than the candidate implementations in the new search space. The first situation indicates that the algorithm cannot identify further opportunities for improvement and thus the algorithm stops. In the second situation, the optimization process may have simply reached a local maxima. To prevent it from being stuck in such a situation, the algorithm probabilistically decides whether to continue searching or not (with a high probability to continue).

As-built Critical Path Analysis We next describe the as-built critical path analysis used to direct the generation of new search space. The as-built critical path analysis is based on the *simulated execution graphs* which are generated from the simulated execution information of the implementations. Let $ExeGraph(I)$ be the simulated execution graph of the implementation I. Nodes in $ExeGraph(I)$ represent events in the simulated execution on the cores and 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 on data from the first task. Edge weights indicate how long it takes to execute the task instance or transfer the data.

The compiler analyzes $ExeGraph(I)$ to discover the path with the largest weight from the start of the execution to the end of the execution, which is the as-built critical path. Note that the standard critical path is only meaningful when there exists sufficient computational resources such that all task instances whose data dependencies have been resolved can be executed in parallel. The as-built critical path is computed based on the simulated execution and therefore captures the actual computational bottlenecks.

Generation of Optimized Implementation The as-built critical path may contain edges because a second task had to wait for a core to become available and not because the task needed the data from the previous task. Thus our optimization algorithm uses the data dependence information for task instances on the as-built critical path to attempt to optimize the implementation by moving tasks to different cores.

For each task instance on the as-built critical path of I, the algorithm computes the time when a task invocation's data dependencies are resolved, which is the earliest point when all the parameter objects of the task instance are ready. The optimization algorithm sorts task instances by the data dependence resolution time. Task invocations that have their data dependencies 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 of task instances to attempt to optimize.

If there are task invocations on the as-built critical path with actual start times that are after the time when their data dependencies are resolved, it means that these task instances were delayed because of resource conflicts and not because of data dependencies. Thus if there are spare cores during the interval between the estimated start time and the actual start time of a task instance, the optimization algorithm attempts to shorten the as-built critical path by generating a set of new implementations in which the task instance has been 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 as-built critical path that produce data that the next task on the as-built critical path consumes. It is likely to be more important to carefully schedule key task instances than other tasks as other tasks depend on the data that key tasks produce. The optimization algorithm identifies situations in which a non-key task instance on the as-built critical path delays the invocation of a key task instance. The algorithm 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 3.3 to generate new candidate implementations which redeploy the chosen task invocations on selected cores. If the set of new optimized implementations is too small, the algorithm randomly chooses other task invocations to also optimize. It then iteratively repeats the simulation, evaluation, and optimization process until several iterations fail to yield improvements.

3.6 Dynamic Scheduling

An alternative 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 sophisticate implementations that account for future data dependencies. For example, we have observed that our approach generates implementations of MonteCarlo 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 heterogeneous cores and network topologies by simply extending the simulation to model these factors.

4 Evaluation

We next discuss our evaluation of Bamboo on four benchmarks: Series, a Fourier series benchmark; MonteCarlo, a Monte Carlo simulation; FilterBank, a multi-channel filter bank for signal processing; and Fractal, a Mandelbrot set computation.

We have implemented the Bamboo compiler. The source code for our compiler is available at `http://demsky.eecs.uci.edu/bamboo/`. We executed our benchmarks on a cycle accurate simulator for the MIT RAW processor.

For each benchmark, we generated three versions: a single-core C version for the RAW processor, a single-core Bamboo version, and a 16-core Bamboo version for the RAW processor. We recorded how many clock cycles were taken by each execution. Figure 5 presents the results.

Benchmark	Clock Cycles($10^6 cyc$)			Speedup to 1-Core Bamboo	Speedup to 1-Core C	Overhead of Bamboo
	1-Core C	1-Core Bamboo	16-Core Bamboo			
Series	25.0	26.4	1.8	14.7	13.9	5.6%
MonteCarlo	138.8	191.7	19.0	10.1	7.3	38.1%
FilterBank	71.1	91.2	6.7	13.6	10.6	28.3%
Fractal	36.2	38.4	3.3	11.6	11.0	6.1%

Fig. 5. Speedup of the Benchmarks on 16 cores

4.1 Benchmarks

We evaluated our approach on four benchmarks:

Series The Series benchmark computes Fourier coefficients. We ported it from the Java Grande benchmark suite [4]. The Bamboo version contains two components: the first divides the work and the second computes the coefficients. The compiler generates a layout that deploys the work division component on a single core and deploys the coefficient computation on all of the cores. The generated binary uses a round robin strategy to distribute the computation of the coefficients across 16 cores. We observed a 14.7× speedup of the 16-core Bamboo version compared to the single-core Bamboo version and a 13.9× speedup compared to the single-core C version.
MonteCarlo The MonteCarlo simulation benchmark was ported from the Java Grande benchmark suite [4]. The Bamboo version is organized into four components: the initialization component, a simulation component, a data aggregation component, and a validation component. Bamboo generates an implementation that deploys both the initialization and validation components on a single core and places an instantiation of the simulation component on each core. The objects created in the initialization component are distributed to the simulation components in a round-robin fashion. The speedup of the 16-core Bamboo version is 10.1× compared to the single-core Bamboo version and is 7.3× compared to the single-core C version.

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 4.4.
FilterBank FilterBank is a multi-channel filter bank for multi-rate signal processing. We ported this benchmark from the StreamIt benchmark suite [5]. FilterBank performs a down-sample followed by an up-sample on each channel and then combines the results for all channels. The Bamboo version is composed of four components: the initialization component, the channel processing component, the aggregation component, and the output component. The generated implementation deploys an instantiation of the channel processing component on each core and deploys the initialization component on a single core. The initialization component distributes the channel processing data in a round-robin fashion. The speedup of the 16-core Bamboo version is 13.6× compared to the single-core Bamboo version and is 10.6× compared to the single-core C version.
Fractal Fractal is a Mandelbrot set computation. The Bamboo version is structured as three components: the first creates original image data and divides it into pieces, the second computes which pixels belong to the Mandelbrot set, and the third combines the processed pieces to generate the final image.

The Bamboo compiler generates a layout that deploys an instantiation of computation component on each core and deploys the initialization component on a single core. The initialization component distributes the separated image data in a round-robin fashion. The speedup of the 16-core Bamboo version is 11.6× compared to the single-core Bamboo version and is 11.0× compared to the single-core C version.

4.2 Accuracy of Scheduling Simulator

The accuracy of the high-level scheduling simulator is important as the automatic tuning is based on the scheduling simulation results. To evaluate the accuracy of the scheduling simulator, we compared the estimated execution time for the 16-core Bamboo implementation chosen by the scheduling simulator with the real execution time of the corresponding 16-core binary for each of our benchmarks. Figure 6 presents the results.

The simulation estimates are close to the real execution time. For Series, the estimation for the 16-core Bamboo version is 5.56% less than the real execution time. A closer examination of the profiling data shows that it takes much less time to compute the first coefficient than the other coefficients. We used the average time to estimate execution time, while the actual time depends on the longest time it takes to compute a coefficient. The 6.06% difference in 16-core version of Fractal has a similar cause.

Benchmark	1-Core Bamboo Version			16-Core Bamboo version		
	Clock Cycles($10^6 cyc$)			Clock Cycles($10^6 cyc$)		
	Estimation	Real	Error	Estimation	Real	Error
Series	26.3	26.4	-0.38%	1.7	1.8	-5.56%
MonteCarlo	191.0	191.7	-0.37%	18.3	19.0	-3.68%
FilterBank	91.2	91.2	0%	6.5	6.7	-2.99%
Fractal	38.4	38.4	0%	3.1	3.3	-6.06%

Fig. 6. Accuracy of Scheduling Simulator

4.3 Optimality of Directed Simulated Annealing

We next discuss our evaluation of the directed simulated annealing approach. For each benchmark, we define the original input as $\text{Input}_{\text{original}}$. As the $\text{Input}_{\text{original}}$ is relatively small and it is relatively straightforward to generate a 16-core implementation, we defined a second input $\text{Input}_{\text{double}}$, which contains a workload that is twice as large. We then collected profiling data for $\text{Input}_{\text{double}}$ — $\text{Profile}_{\text{double}}$. We used the Bamboo compiler to synthesize 16-core implementations for each benchmark with the $\text{Profile}_{\text{double}}$ to evaluate the likelihood that the directed simulated annealing algorithm will generate a good implementation.

For each benchmark, we first generated all possible candidate implementations and used the scheduling simulator to evaluate them. The empty bars in Figures 7, 8, 9, and 10 present the probability distributions for the candidate implementations. The x-axis is the estimated executing time of the candidate implementations. The y-axis is the relative percentage of the particular estimated execution time. Candidate implementations with smaller estimated execution times are 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 Figures 7, 8, 9, and 10 presents the probability distribution 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. In all cases, it took less than 9.0 seconds for the directed simulated annealing algorithm to find the best implementation.

4.4 Generality of Synthesized Implementation

We used profiling data to generate an implementation that is optimized for the target multi-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.

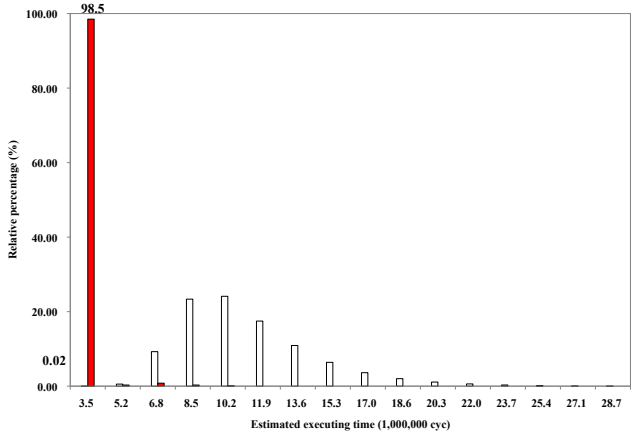

Fig. 7. Candidate Distribution for Series

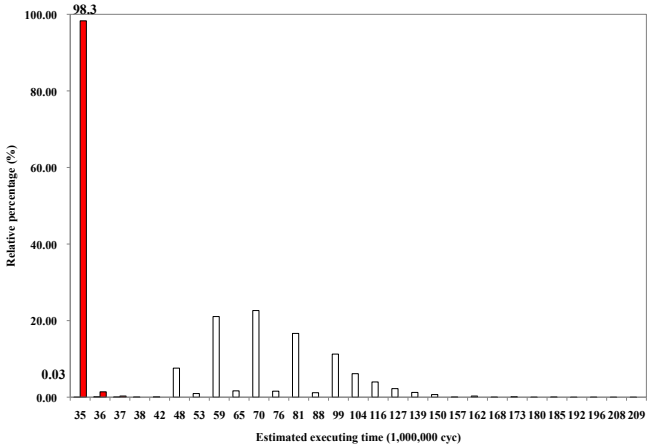

Fig. 8. Candidate Distribution for MonteCarlo

We next discuss our evaluation of how well our optimized implementation generalized to other inputs. For each benchmark, we generated a new 16-core Bamboo version using the $\text{Profile}_{\text{double}}$ described in the previous section. We executed the new version as well as the single-core Bamboo version and the 16-core Bamboo version generated with $\text{Profile}_{\text{original}}$ on the new $\text{Input}_{\text{double}}$. Figures 11 presents the results.

For most benchmarks, the speedup of both 16-core Bamboo versions are similar indicating that the synthesized binaries generalize to different inputs. For MonteCarlo, the 16-core version generated using $\text{Profile}_{\text{double}}$ performs better on $\text{Input}_{\text{double}}$ than the 16-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. In this case the acceleration is small because that the overlapped simulation computation

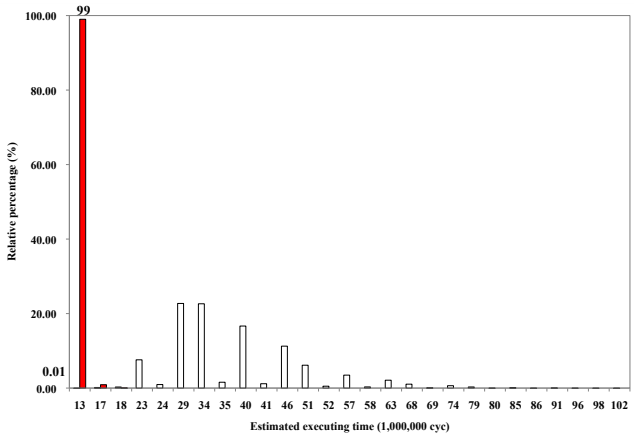

Fig. 9. Candidate Distribution for Filterbank

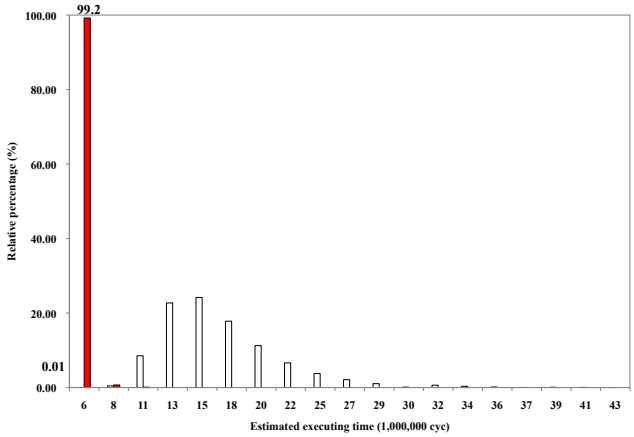

Fig. 10. Candidate Distribution for Fractal

is relatively lightweight. However, it shows that the compiler successfully generated a sophisticated parallel implementation.

4.5 Overhead of Bamboo

We next compare the performance of the single-core C version and the single-core Bamboo version for each benchmark to characterize the overhead of Bamboo. We found that for Series and Fractal, the overheads of the Bamboo language, including cache flushes and runtime scheduling, are 5.6% and 6.1% respectively. For FilterBank and MonteCarlo, the overheads are 28.3% and 38.1%, respectively. After a further investigation, we found that the primary cause of these overheads is that the GCC compiler does not reorder floating point operations in the intermediate code generated by Bamboo compiler as it does on the C code. Because the RAW chip does not implement out of order execution, it introduces stalls when two floating point operations are issued back to back. As the code Bamboo generates uses array objects instead of the stack allocated

Benchmark	$\text{Profile}_{\text{original}}$, $\text{Input}_{\text{double}}$			$\text{Profile}_{\text{double}}$, $\text{Input}_{\text{double}}$	
	Clock Cycles$(10^6 cyc)$		Speedup	Clock Cycles$(10^6 cyc)$	Speedup
	1-Core	16-Core		16-Core	
Series	54.2	3.6	15.1	3.6	15.1
MonteCarlo	383.2	37.8	10.1	35.7	10.7
FilterBank	182.3	13.3	13.7	13.3	13.7
Fractal	76.6	6.5	11.8	6.5	11.8

Fig. 11. Generality of Synthesized Implementations

arrays used in the C version, GCC's alias analysis is not able to determine that writes to the array objects does not change the values of stack allocated variables.

As the necessary alias information is implied by Bamboo's types, a commercial Bamboo compiler would not incur these overheads. Moreover, modern out-of-order processors are less sensitive to the exact ordering of instructions. We therefore expect that the performance of a commercial Bamboo implementation would be similar to C.

5 Related Work

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 [6]. Bamboo borrows ideas from dataflow and integrates them within the context of a standard imperative language to ease adoption by developers. Bamboo relaxes typical 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.

Tuple-space languages, such as Linda [7], 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.

Orc [8] specifies how work flows between tasks. Another language, Oz, is a concurrent, functional language that organizes computations as a set of tasks [9]. Bamboo's computational model is similar to actors. Actors communicate through messages [10]. We note that because actors (unlike tasks) contain state, an individual actor is not straightforward to parallelize. Bamboo borrows basic language constructs from Bristlecone [11], but extends the previous work to support parallel execution.

Streaming languages [5] 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.

Recently, self-tuning libraries such as PhiPAC [12], FFTW [13] and SPIRAL [14] have leveraged similar empirical search-based approaches for automatic optimization. However, Bamboo targets general computation while these approaches address specific computations.

6 Conclusion

Automatic tuning can be an effective technique for optimizing applications for multi-core processors. A developer using our approach simply writes their program using our task-based extensions to Java. The Bamboo compiler combines information from the task declarations, profiling information, and a processor description to generate implementations that are tuned for a specific multi-core processor. The tuning algorithm was able to generate sophisticated implementations that scaled successfully to 16 cores. These implementations generalized to inputs that were different from the original profiling inputs. We expect that our automatic tuning technique can be extended to the broader context of traditional programming languages. The idea is to extract dependence information from executions and then use our tuning framework to provide guidance to developers about how to best refactor their code for a parallel implementation.

Acknowledgements

This research was supported the National Science Foundation under Grant No. CCF-0725350.

References

1. Jenista, J., Demsky, B.: Disjointness analysis for Java-like languages. Technical Report UCI-ISR-09-1, Institute for Software Research, University of California, Irvine (February 2009)
2. Demsky, B., Sundaramurthy, S.: Static analysis of task interactions in bristlecone for progra m understanding. Technical Report UCI-ISR-07-7, Institute for Software Research, University of California, Irvine (October 2007)
3. Larson, H.J., Shubert, B.O.: Probabilistic Models in Engineering Sciences. Wiley (1979)
4. Smith, L.A., Bull, J.M., Obdrzalek, J.: A parallel Java Grande benchmark suite. In: Proceedings of SC2001. (2001)
5. Gordon, M. et al.: A Stream Compiler for Communication-Exposed Architectures. In: International Conference on Architectural Support for Programming Languages and Operating Systems. (October, 2002)
6. Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in dataflow programming languages. ACM Comput. Surv. **36**(1) (2004)
7. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems (TOPLAS) **7**(1) (1985) 80–112
8. Cook, W.R., Patwardhan, S., Misra, J.: Workflow patterns in Orc. In: Proceedings of the 2006 International Conference on Coordination Models and Languages. (2006)
9. Smolka, G.: The Oz programming model. In: Proceedings of the European Workshop on Logics in Artificial Intelligence, London, UK, Springer-Verlag (1996) 251
10. Hewitt, C., Baker, H.G.: Actors and continuous functionals. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA (1978)
11. Demsky, B., Dash, A.: Bristlecone: A language for robust software systems. In: Proceedings of the 2008 European Conference on Object-Oriented Programming. (2008)
12. Bilmes, J., Asanovic, K., Chin, C.W., Demmel, J.: Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In: Proceedings of the ACM International Conference on Supercomputing. (1997) 340–347
13. Frigo, M.: A fast Fourier transform compiler. In: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation. (1999) 169–180
14. Püschel, M. et al.: SPIRAL: Code generation for DSP transforms. Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" **93**(2) (2005) 232–275