Trace-driven Simulation of Multithreaded Applications

Alejandro Rico*, Alejandro Duran*, Felipe Cabarcas*‡, Yoav Etsion*, Alex Ramirez*† and Mateo Valero*†
*Barcelona Supercomputing Center
Centro Nacional de Supercomputacion, Barcelona, Spain
Email: {first.last}@bsc.es
†Universitat Politecnica de Catalunya, Barcelona, Spain
‡Universidad de Antioquia, Medellín, Colombia

***Abstract*—Over the past few years, computer architecture research has moved towards execution-driven simulation, due to the inability of traces to capture timing-dependent thread execution interleaving. However, trace-driven simulation has many advantages over execution-driven that are being missed in multithreaded application simulations.**

We present a methodology to properly simulate multithreaded applications using trace-driven environments. We distinguish the intrinsic application behavior from the computation for managing parallelism. Application traces capture the intrinsic behavior in the sections of code that are independent from the dynamic multithreaded nature, and the points where parallelism-management computation occurs.

The simulation framework is composed of a trace-driven simulation engine and a dynamic-behavior component that implements the parallelism-management operations for the application. Then, at simulation time, these operations are reproduced by invoking their implementation in the dynamic-behavior component. The decisions made by these operations are based on the simulated architecture, allowing to dynamically reschedule sections of code taken from the trace to the target simulated components.

As the captured sections of code are independent from the parallel state of the application, they can be simulated on the trace-driven engine, while the parallelism-management operations, that require to be re-executed, are carried out by the execution-driven component, thus achieving the best of both trace- and execution-driven worlds.

This simulation methodology creates several new research opportunities, including research on scheduling and other parallelism-management techniques for future architectures, and hardware support for programming models.

I. Introduction

Computer architecture research is mainly based on architecture simulation, since the execution of a workload on a processor chip can hardly be modeled analytically, and prototyping every design point to be evaluated is prohibitively expensive. Thus, software simulators are fundamental to explore architecture design options.

The simulation of parallel architectures is even more challenging than simulating a single processor, since several execution streams stress not only processor-private components but also shared resources in the architecture (caches, interconnection network and memory). Modeling such sharing and resource contention is difficult and is very sensitive to the level of parallelism, inter-core data sharing and thread synchronization of the particular application. In this paper we refer to the architecture being simulated as the *target*, and the machine on which the simulator runs as the *host*.

For simulating single-cores, and multi-core architectures running multiprogrammed workloads, the use of trace-driven simulators offers some significant advantages over execution-driven tools. By employing traces, researchers get more flexibility to perform their simulation experiments because they get rid of the computational requirements of the workload running on the target architecture. Contrarily, on execution-driven simulations, the workload system requirements (such as memory size or disk space for the input data set) may easily exceed the host computational resources specially for large-scale multi-core simulation. As an example, some high-performance applications require several hundred *megabytes* of allocated data per thread [1]. When exploring future many-core architectures with hundreds of cores, this may result in tens of *gigabytes* of memory, which limits the execution-driven simulation of such scenarios to large-scale host machines.

In some situations, the access to target application codes and binaries is restricted, so execution-driven simulation is just not possible. In these cases, traces can usually be distributed instead, thus allowing researchers to work with the applications of interest regardless of the actual access to the application binaries.

Moreover, execution-driven simulation implies a higher modeling complexity, such as situations where the endianness of the target ISA and the host differ. Techniques such as *native execution* also limit the kind of host machines that can be used for execution-driven simulation. Traces are, in this sense, ISA independent and easier to simulate in a larger variety of host machines.

Simulation based on traces is also potentially faster than execution-driven: it does not require to move and copy application data, but rather work just with tags. Additional speed can also be provided by raising the level of abstraction because simulation is not tied to instruction-level operation. Discrete Event Simulation [2] provides large speed improvements at the expense of loss of accuracy, which is very useful for some particular uses (e.g., early design decisions; memory system and network evaluations) and is only usable in trace-driven environments.

978-1-61284-368-1/11/$26.00 ©2011 IEEE

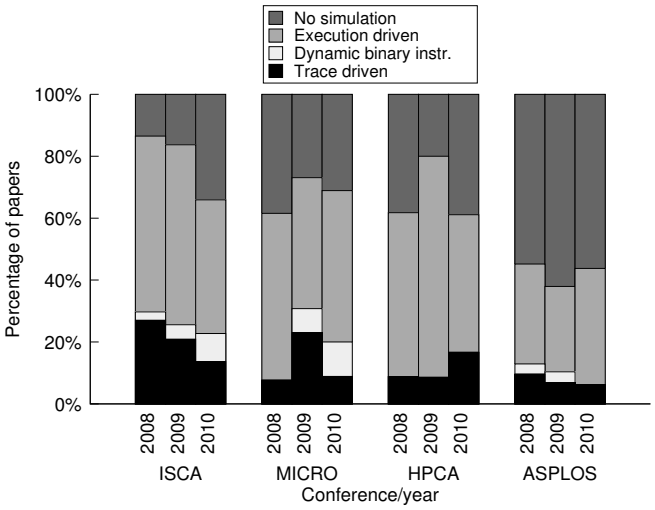


Fig. 1. Simulation types in computer architecture conferences.

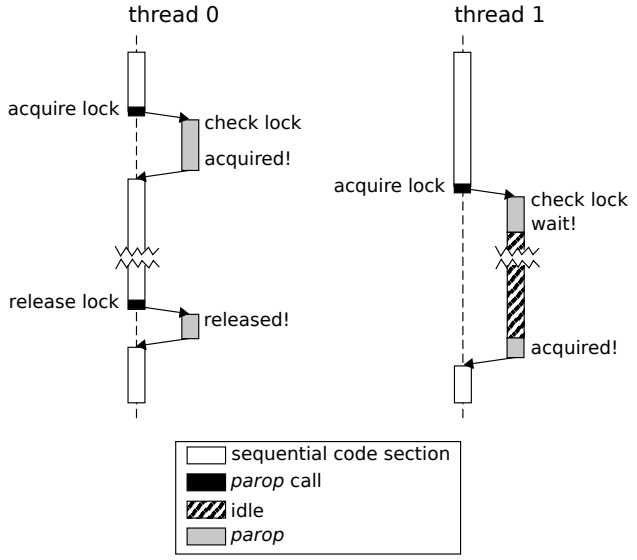


Fig. 2. Execution of an application with a mutual exclusion using two threads.

However, over the last few years, multi-core research has moved towards execution-driven simulation. Figure 1 shows the simulation type used in the papers from four of the most important computer architecture conferences since 2008. Execution-driven tools clearly dominate among papers that include simulation-based evaluations. This is primarily due to the inability of traces to capture the timing-dependent execution interleaving of multithreaded applications. This includes dynamic thread scheduling and inter-thread synchronizations that determine the order in which parallel code sections are executed in different threads. In this sense, whenever the target architecture changes due to the evaluation of different design choices (core microarchitecture, caches, interconnection network or memory configurations), the execution-driven simulation of multithreaded applications will vary accordingly providing the desired insight. Contrarily, trace-driven simulations will not modify the overlap and order of the different thread execution streams. Additionally, traces collected on a specific number of cores limit the simulation to that same number of cores, forcing to generate traces for each number of cores to be evaluated.

In this paper we propose a simulation methodology that combines trace-driven simulation with the dynamic execution of the parallelism-management operations in the application, to benefit from the best of both trace- and execution-driven worlds. This results in flexible trace-based simulations with no limitations for the proper evaluation of multithreaded applications. In the proposed methodology, the tracing engine distinguishes the code that is independent from the dynamic nature of multithreaded applications, from parallelism-management operations (*parops*[1]), such as scheduling or synchronization, which depend on the parallel state of the application at a given execution point. Thus, the instrumentation of application with such recognizes and captures *parop* calls at runtime, instead of blindly recording the instruction stream, which would prevent untying the simulation from the statically captured thread scheduling. The captured *parops* are reproduced at simulation time to perform the suitable scheduling decisions (as if these were executed on a real machine), thus allowing the dynamic scheduling of pieces of data and computation and its assignment to different simulated threads. All in all, different target architecture configurations can be properly evaluated with a single trace per application, which dramatically reduces tracing time, while taking advantage of the aforementioned benefits of trace-driven execution.

[1]For simplicity, we abbreviate parallelism-management operations with *parops* in the rest of this paper.

II. Methodology

The novel methodology proposed in this paper allows to properly simulate parallel architectures running multithreaded applications using a trace-driven approach. For this purpose, we distinguish the application intrinsic behavior from the parallelism-related computation. The application intrinsic behavior is independent of its dynamic multithreaded nature, and it is the same in a sequential machine or in a parallel system. The computation related to parallelism management determines how the application computation is distributed among threads and how different execution streams interleave. These operations that manage parallel work (*parops*) depend on the features and availability of the resources in the execution environment.

The application intrinsic behavior is represented by its *sequential code sections*. These are instruction stream regions where execution order remains unaltered regardless of any scheduling and synchronization occurrences. The order of the instructions in a sequential code section is unmodifiable, meaning that instruction i (in sequential order) will always execute

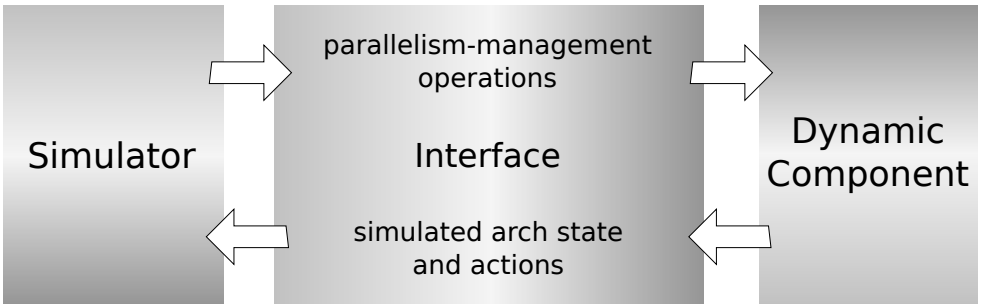


Fig. 3. Simulator – dynamic component interface.

after instruction *i-1* and before instruction *i+1*. Examples of sequential code sections are an iteration in an OpenMP parallel loop, or a Cilk [3] parallel routine.

Parops determine the order and timing in which sequential code sections are executed. These include, but are not limited to: thread management (creation/spawn/join), synchronizations (e.g., locks, waits), parallel work scheduling, code/data transfers (in distributed memory machines), and transaction begin and commit (in transactional memory systems).

Figure 2 shows the execution of a multithreaded application with a mutual exclusion. Dashed lines show the *sequential code sections* (one per thread), which are separated in several fragments shown as white boxes. The computation outside the dashed lines are *parops* (in the example, acquiring/releasing a lock), which are shown in gray. The small black boxes show the calls to these operations. Thread 0 starts executing a sequential code section until it reaches the point where it needs to acquire a lock to enter the mutual exclusion. At that point, the operation to acquire the lock executes, it actually gets the lock, and thread 0 enters the mutual exclusion. Thread 1 tries to acquire the lock but it has to wait for thread 0 to release it. When thread 0 finishes the mutual exclusion code, it releases the lock. Then, thread 1 acquires it, and enters the mutual exclusion.

As it can be noticed in the figure, the execution of a sequential code section can interleave with the execution of *parops*, and that does not break its sequentiality property. Also, the execution order of a sequential code section does not depend on the state of parallelism-related data structures or the available computing resources. Contrarily, the execution of the lock acquire in the figure actually varied (from thread 0 to thread 1) depending on the status of the lock.

Given these properties, our methodology establishes that sequential code sections (including their *parop* calls) are captured using a trace-driven approach; and that *parops* are dynamically re-executed at simulation time using an execution-driven approach. Therefore, application traces include the sequential code sections of the application with the corresponding *parop* calls (shown in white and black in Figure 2). The execution of sequential code sections is represented in traces through the list of executed instructions. However, the traces in our methodology do not include the execution of *parops* themselves (shown in grey in Figure 2). Instead, they include the information related to the execution of *parops*, so *parop* calls can be reproduced at simulation time. Also, application traces are indexed, so separately-captured sections of code can be accessed individually. This is necessary since the execution of a *parop* may cause a change in the control flow, so the calling thread may switch to run a new sequential code section, or resume the execution of a suspended one. In the example in Figure 2, the corresponding trace would include the two sequential code sections (the ones originally executed in thread 0 and thread 1), the acquire/release lock calls, and the address of the lock for each call.

Using this tracing method, an application is fully represented in a single trace, and it is not necessary to generate a trace for every multithreaded configuration (e.g., different number of cores), which results in a dramatic reduction in the required tracing effort and time.

Once traces are generated, the simulation tool needs to have access to an implementation of the required *parops*. *Parops* are typically provided as part of a runtime system library or a specific thread library, which is referred to as the *dynamic component* in this methodology. In order to work as decoupled as possible, a clearly-defined interface is required to provide the necessary services to both the simulator and the dynamic component.

Figure 3 shows a conceptual scheme of the simulator – dynamic component integration. From one side, the dynamic component has to expose all *parops* to the simulator, so they can be accessed at any point in simulation time. From the other side, the simulator has to expose the target architecture state to the dynamic component, so it can be checked as needed for *parops* to work properly. For this purpose, the architecture-dependent functionalities in the dynamic component have to be replaced by specific functionalities for the target architecture. The effort required for this replacement heavily depends on the level of encapsulation of the architecture-dependent functionalities in the dynamic component implementation.

Multithreaded application simulations using the described traces and the simulation platform, which includes the simulation engine and the dynamic component, run as follows. The simulator begins running the starting thread in the application, commonly known as the *main* or *master* thread. This main

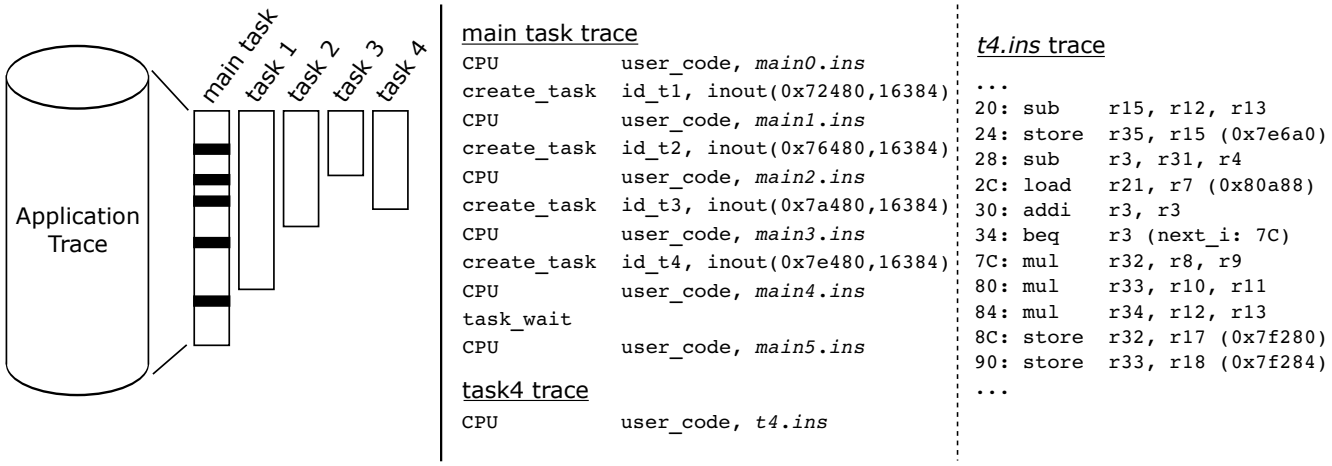


Fig. 4. Overview of traces for the TaskSim – Nanos++ integrated platform. The structure in the figure corresponds to the source code in Listing 1. Each task has its own trace section. Computation between parops are `CPU` events with an associated instruction trace containing the corresponding executed instructions.

thread is represented with the corresponding sequential code section. The simulator runs sequentially until it reaches the first *parop* call in the trace. At that point, the simulator summons the dynamic component by calling the corresponding *parop* through the intermediate interface. The corresponding *parop* is dynamically executed at simulation time as if it was called during the execution on a real machine. The operation carries out its functionality depending on the characteristics of the target simulated machine (which may be completely different from the host architecture where it is running), instead of depending on the characteristics of the machine where the trace was captured (as it happens on traditional statically-simulated trace-driven tools). For example, scheduling decisions performed by the dynamic component are made based on the number of cores of the target simulated architecture, the core status (running/wait/idle), or the core work load at a given point in simulation time.

After dynamically executing a given *parop*, its timing effects are simulated and the different threads in the target architecture may eventually have sequential code sections scheduled, according to the dynamic component decisions. Execution in different threads is then simulated by assigning the suitable sequential code sections in the trace to the corresponding simulated threads. For instance, when a parallel Cilk function is invoked, the corresponding *parop* is called, which creates and executes the corresponding parallel work. At the same time, the sequential code section that invoked the function is scheduled to another thread to continue its execution. When the *parop* call returns, one of the threads will have the invoking code section scheduled, which is simulated by assigning that sequential code section to the corresponding thread in the target architecture. This operation is possible because different sequential code sections are captured in the trace separately and can thus be independently executed on different threads in the target architecture.

This methodology works for any *parop* as long as the target architecture state is exposed, as necessary, to the architecture-dependent part of the dynamic component. Altogether, the proposed methodology achieves the simulation of multithreaded applications on parallel architectures, with a combination of trace-driven simulation and dynamic execution of *parops*.

```
#pragma css task inout(mat)
void t1( float mat[N][N] );

#pragma css task inout(mat)
void t2( float mat[N][N] );

#pragma css task inout(mat)
void t3( float mat[N][N] );

#pragma css task inout(mat)
void t4( float mat[N][N] );
```

```
float matrix[4][N][N];

int main( int argc, char* argv[] )
{
    //...
    t1( matrix[0] );
    //...
    t2( matrix[1] );
    //...
    t3( matrix[2] );
    //...
    t4( matrix[3] );
    //...
    #pragma css wait

    //print matrix contents
}
```

Listing 1. StarSs code example.

III. Sample Implementation

We have implemented the proposed methodology using TaskSim [4], a trace-driven simulator for large-scale architectures, and the Nanos++ runtime system [5]. The implementa-

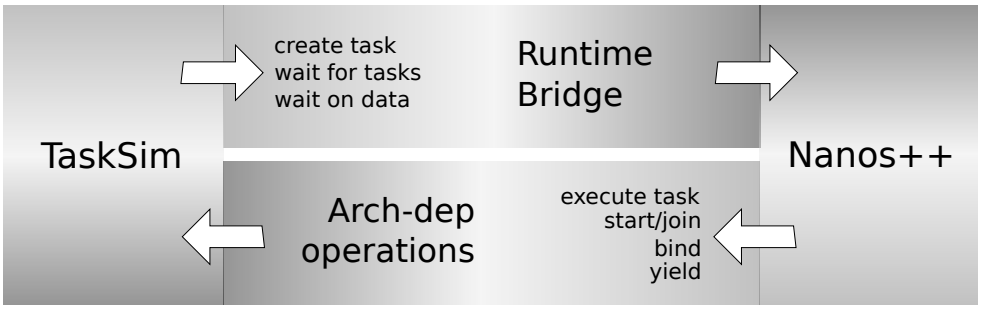


Fig. 5. Interface scheme between TaskSim (the simulator) and Nanos++ (the dynamic component). The Runtime Bridge encapsulates the communication and gives TaskSim access to Nanos++ operations. The architecture-dependent part provides Nanos++ access to the simulated architecture.

tion supports multithreaded applications written in any of the programming models supported by Nanos++, which currently include OpenMP 3.0 and StarSs (CellSs [6] and SMPSs [7] are implementations of StarSs for different platforms).

The StarSs parallel programming model allows the definition of potentially parallel tasks on a sequential code using pragma annotations on functions. The task input and output data are similarly defined using annotations on function arguments, which allows the automatic resolution of task dependencies and extracting task parallelism at runtime. The Mercurium source-to-source compiler [8] implements the necessary transformations to compile OpenMP 3.0 and StarSs applications for Nanos++.

Listing 1 shows a basic example of a StarSs code. Functions `t1`, `t2`, `t3` and `t4` are annotated as tasks, all of them having their paramater `mat` defined as input-output data. At each function call point during `main` execution, the corresponding parallel tasks are automatically created and scheduled accordingly. Later, the `css wait` clause forces the `main` execution to synchronize with the previously-created tasks before printing the matrix contents.

Having this environment, the target applications are first compiled using Mercurium. The resulting binary uses the Nanos++ runtime system library, which is instrumented to generate suitable traces for TaskSim. The resulting application traces are simulated on TaskSim, and the Nanos++ runtime system library is used as the dynamic component for the execution of *parops* at simulation time.

A. Instrumentation and Interface

Nanos++ incorporates an instrumentation engine that can be used to generate traces with a specific trace format via the implementation of plugins that are dynamically loaded at runtime. For our implementation, we developed a Nanos++ plugin to capture the runtime events of interest and generate traces in the TaskSim trace format. Additionally to the captured events, instruction traces are generated using PIN [9]. Since all *parops* are clearly encapsulated in the runtime system, the instrumentation captures all the necessary information for the proposed methodology.

The sequential code sections in this environment correspond directly to Nanos++ tasks. The main execution thread, which is a sequential code section in itself, is treated as a Nanos++ task as well. In the rest of this section, we refer to the sequential code sections as *tasks*, and to the main-thread associated task as *main task*.

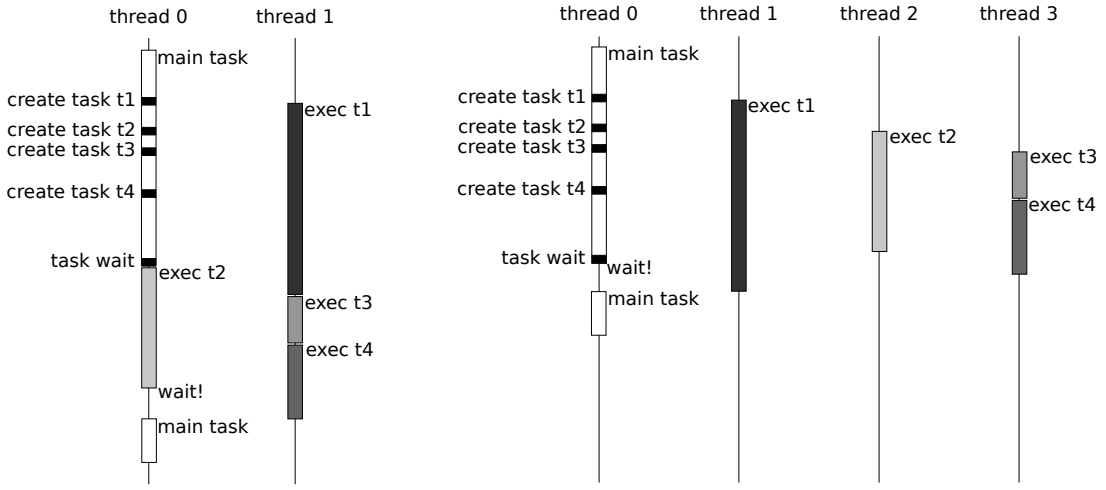
Figure 4 shows a scheme of the resulting traces from the application example in Listing 1. Task traces are captured separately and include all *parop*-call events as they originally occurred. The *application trace* file contains all task traces, and they can be accessed individually through an index. The information related to each *parop* call (function arguments) is also included in the trace, so they can be properly re-executed at simulation time. Also, computation periods between *parops* are specified as `CPU` events and include the name of the associated instruction trace file.

Since neither StarSs nor OpenMP allow user-defined *parops*, the Nanos++ runtime system encapsulates all necessary operations. This allows the use of the Nanos++ library as the dynamic component to be integrated with TaskSim without any modifications in the core functionalities, only requiring to add the architecture-dependent part for the target architecture.

Once both Nanos++ and TaskSim are integrated, the runtime system operations are exposed to the simulation engine through an interface referred to as the *runtime bridge*. Whenever the simulation in TaskSim finds a *parop*, the corresponding Nanos++ service is invoked through the runtime bridge. Similarly, when Nanos++ needs a simulated component in TaskSim to perform a specific action, it is requested through its architecture-dependent part.

Figure 5 depicts the described structure with the main instrumented *parops*, and the architecture state information and actions exposed to Nanos++. The main *parops* in this implementation (and the corresponding function names in the Nanos++ library) are detailed below along with the necessary captured information for their reproduction at simulation time:

- create task (createWD): the calling task requests the creation of a new task. It specifies the new task id and the input and output data identifiers (address and size) for dependence tracking. The new task may be submitted for execution in any thread, or it can be executed *inline* (immediately executed in the requesting-thread execution stream).



(a) 2 cores (b) 4 cores

Fig. 6. Scheme of a multithreaded application execution that generates 4 tasks and waits for their completion on 2 and 4 cores.

- task wait (waitCompletion): the calling task requests to wait for all its previously created tasks to complete execution. No additional information is required.
- wait on data (waitOn): the calling task requests to wait for the last producer of a specific piece of data to complete execution. It specifies the corresponding data identifiers.

These operations cover the basic functionality of StarSs applications and the tasking support of OpenMP 3.0. Additionally, there are available operations for finer-grain synchronization semantics, such as setting/checking synchronization variables (e.g., semaphores), or acquiring/releasing locks.

Similarly, TaskSim exposes the architecture state to Nanos++ through the architecture-dependent part. These architecture-dependent operations are able to assign tasks to simulated threads and set their state (idle/running). The architecture-dependent part has an interface to the Nanos++ runtime system core, which is common for any underlying architecture, and includes the following operations:

- execute task (switchTo/inlineWork): execute a task on the current thread. It can be done on a separate context (switchTo) or on the current context (inlineWork).
- start/join thread (start/join): start/finishes a new thread.
- bind thread (bind): bind the current thread to a hardware CPU thread.
- yield thread (yield): requests a thread to yield.

Additionally, the architecture-dependent part provides support for explicit data transfers on distributed shared memory architectures, such as systems including local/scratchpad non-coherent memories (e.g., GPU, multimedia, Cell/B.E.).

For the aforementioned functions to work properly, a set of data structures representing the several threads at the dynamic-component layer need to be created on startup. The simulator engine calls the dynamic component to initialize these data structures through the *init* runtime-bridge function with the suitable target architecture features, such as the number of cores, core types (in heterogeneous scenarios) or the size of local memories (in distributed memory architectures). At this point, the dynamic component starts and initializes the generic data structures that are common for any underlying architecture. Then, in order to initialize the architecture-dependent data structures, it invokes the implementation of the *createThread* routine in the architecture-dependent part.

This pre-simulation procedure provides the dynamic component with the necessary abstraction layer to represent the threads in the target architecture. Thanks to this fact, and the decoupling between the *parop* library and the application code, no data needs to be stored in the trace; and, as a result, the data structures resulting from the runtime system initialization are sufficient for the correct functioning of the previously-listed operations at simulation time.

B. Simulation Example

TaskSim simulation starts by assigning the main task to one of the simulated threads. All other simulated threads start *idle*. A separate context (user-level thread) is created for each simulated thread, to provide the Nanos++ runtime system with the illusion of a multithreaded machine, even though TaskSim is running on a single thread. Then, whenever a simulated thread has to call a *parop*, the simulation switches to the context associated to that simulated thread before entering the runtime system execution.

Figure 6 shows a sample simulation of the application example in Listing 1, with two (Fig. 6(a)) and four cores (Fig. 6(b)). Having all simulated threads setup, the main task runs until it encounters the first `create task`. Then, it calls the corresponding function in the runtime bridge, which summons the creation of task `t1` in the Nanos++ library. Task `t1` is then available for execution, and thread 1, which is idle, starts executing it. Later, the main task also creates `t2`, `t3` and `t4`. These are executed by different threads depending on the available cores in both examples, following a *breadth first*

scheduling policy. After creating the four tasks, the main task simulation finds a `task wait`. At that point in the example with four cores, all created tasks have already started running, and thread 0 just waits for their completion. In the example with two cores, only `t1` has started execution, so thread 0 starts `t2`. Thread 1 finishes `t1`, and then executes `t3` and `t4`. Thread 0 completes `t2`, and switches back to the main thread where it waits `t4` to complete.

This example shows that several simulations with different hardware configurations are possible using a single trace with five tasks (four + *main*) and the appropriate *parops* in place.

C. Experiments

This section presents a set of experiments that show the potential of the presented methodology, using the TaskSim – Nanos++ implementation.

The first experiment is the evaluation of the scalability of a target StarSs application. We use a block-partitioned matrix multiplication where each matrix block is processed by a StarSs task. The BLAS library [10] functions are employed to perform the computation. Figure 7 shows the scalability (speed-up with respect to the execution with 8 cores) of the described matrix multiplication, labeled as *matmul*.

As it can be seen, matmul performance does not scale beyond 32 cores. The analysis of the experiment results showed that the bottleneck is the task generation rate in the main task, a limitation that has been studied in previous works [11]. This issue leads to major core underutilization, as it is shown for the 64-core experiment in Figure 8(a). Each row in the figure corresponds to one core: gray-colored horizontal bars show computation phases and white regions are idle time. The first row at the top of the figure is the main task, which is the only solid line in the figure. Although the main task is continuously busy creating tasks (shown in black), the execution of task creation operations is not fast enough to generate tasks at a sufficient rate to feed all 63 remaining cores. This results in long idle periods (in white) where cores are waiting for tasks.

We modified the application to parallelize the task generation code using a hierarchical task spawn scheme: the main task creates a set of task-spawn intermediate tasks, each of them creating a subset of the total matrix multiplication tasks. Figure 7 shows that the matrix multiplication version with hierarchical task spawn (labeled as *matmul-par*) perfectly scales up to 128 cores. Figure 8(b) shows the core activity for matmul-par. All core rows show solid lines, representing that all cores are continuously busy (no idle periods), thus fully utilizing the available resources in the architecture. The black regions showing the task generation phases are now spread across all cores.

This first experiment shows that the methodology allows trace-driven simulations of multithreaded applications for different number of cores using a single trace per application. Also, thanks to the simulation engine capabilities, multithreaded applications can be evaluated for large numbers of cores, which are not easily available in real machines.

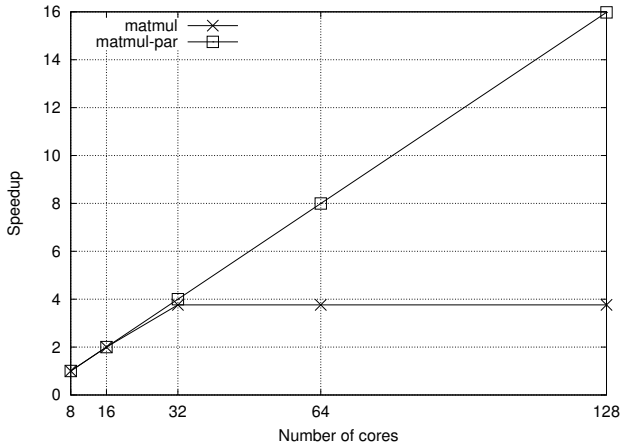


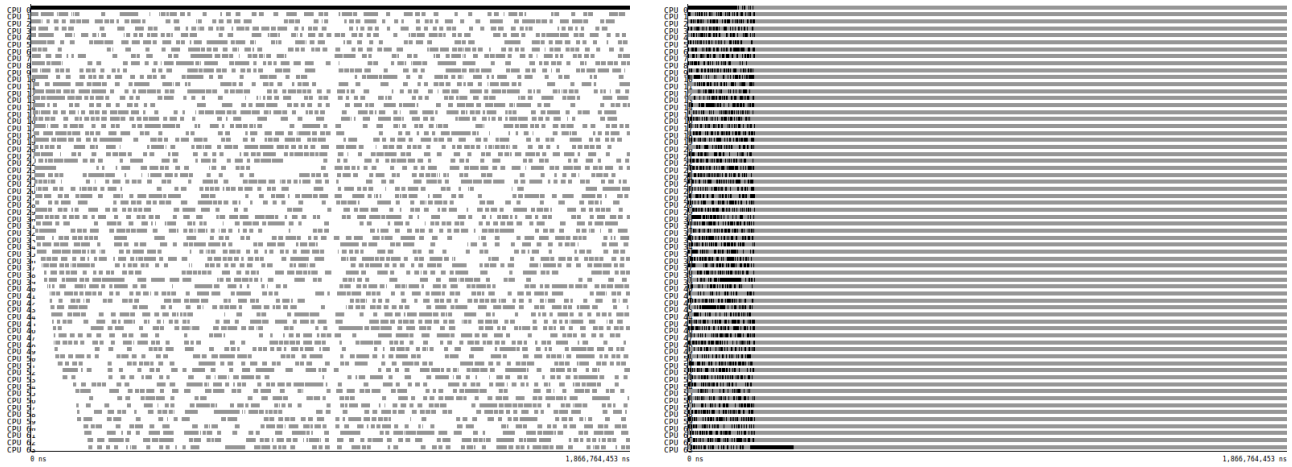
Fig. 7. Speed-up for architecture configurations with different number of cores compared to the execution with 8 cores.

The second experiment is the comparison of different area-equivalent multi-core designs. The different machine configurations follow Pollack's Rule [12], in which the performance increase of a core design is roughly proportional to the square root of the increase in complexity (i.e., core area). The baseline configuration in this experiment includes 16 cores. Then we explore two alternatives: one with four cores twice as fast, and another containing 64 cores with half the performance each.

Figure 9 shows the speed-up with respect to the baseline configuration using 16 cores. The results in the figure are shown for a set of scientific applications programmed using the StarSs programming model. A single trace per application is used for the simulation of all hardware configurations. The configuration with four cores gets better performance for applications with less parallelism, as individual tasks will execute faster and they do not get any benefit from having more cores available. This is the case of *pbpi* and *stap*, which are limited by the performance of the main task, and they do not scale to 16 cores. In this case, doubling the core performance, doubles the overall application performance, as the sequential bottleneck that dominates execution is completed twice as fast.

The configuration using 64 cores performs better for highly-parallel applications, as the benefit of executing more tasks in parallel compensates the individual task performance loss. In our experiments, this applies to applications that scale up to 64 cores, which is the case of *matmul-par*. For the rest of applications, the baseline configuration is the best option. *Cholesky*, *Kmeans* and *matmul* scale to 16 cores, so this configuration outperforms the one using just four cores twice as fast. However, they do not scale to 64 cores, thus having an overall performance loss due to the lower throughput of the individual cores.

This experiment shows that several architecture configurations varying the number of cores, and even the core type/performance are doable using a single trace for each application.



(a) matmul (b) matmul-par

Fig. 8. Snapshot of the core activity visualization of a blocked-matrix multiplication (a) with 1 task-generation thread, and (b) with a hierarchical task-generation scheme

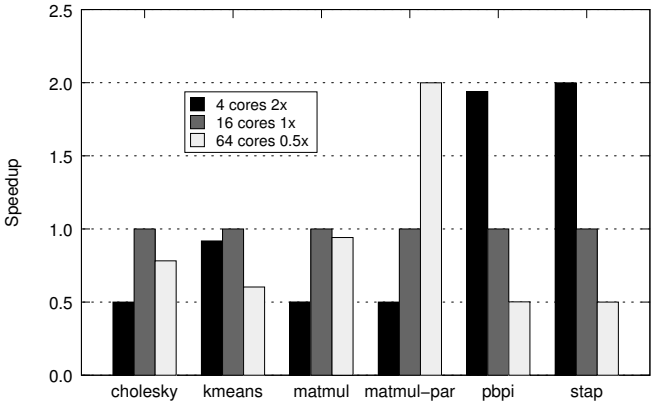


Fig. 9. Application performance comparison for area-equivalent multi-core configurations. Speed-up with respect to the 16-core baseline.

IV. Coverage Discussion and Opportunities

The proposed methodology covers multithreaded application that can be properly instrumented and parallelism-management code is decoupled from the application actual computation. Low-level programming models (such as *pthreads*) allow the user to embed parallelism-management code (such as scheduling and synchronization) in the user application code. Such applications would have to be modified so that parallelism-management code is decoupled from user code, and *parops* can be exposed to the simulator. This would have to be done for each *parop* and every application which is actually impractical. This results in a limitation of the proposed methodology, that is directly inherited from the mentioned weakness of low-level programming models.

Also, non-deterministic applications like those which computation rely on *random* values (e.g., Monte Carlo algorithms) or those including race conditions, may not be reproducible for simulation. All required computation paths may have not executed on the original run, and thus are not available in the trace, making its trace-driven simulation not doable.

Therefore, we have identified high-level programming models (such as OpenMP, Cilk [3], Sequoia [13], StarSs [6], [7], Intel TBB [14] or IBM X10 [15]) as the best cases for our methodology. As *parops* are fully provided in the associated runtime system library, the user application code is completely decoupled from the parallelism-management services. Then, the instrumentation of the runtime system library covers the generation of traces for any application in that programming model. Most of these programming models allow the specification of parallel work as *tasks*, being the instrumentation of separate sequential pieces of code very simple (as it has been found in the sample implementation presented in Section III). Furthermore, the runtime system libraries can be directly used as the dynamic component in the simulation platform, just by adding the corresponding target-architecture-dependent part.

This simulation methodology opens several research opportunities that are not practical using state-of-the-art trace-driven simulation.

First of all, new ideas on runtime system design, implementation and optimization, such as new scheduling policies or more efficient management of thread/task data structures, can be evaluated on future parallel architecture designs before the actual hardware is available. These evaluations are possible thanks to the interface described in Section II, which provides runtime systems with the illusion of being executed on a real machine. Machines with large numbers of homogeneous cores (64-128) accessing a shared memory are nowadays available, although their cost is a limiting factor for its accessibility. The presented framework allows runtime system development for such large-scale machines.

An interesting area of research is scheduling for heteroge-

neous architectures. Nowadays, such studies are being performed in configurations with a general-purpose chip plus a graphics card attached to a PCIe slot. The large penalty for off-chip transfers to the graphics card memory sets some scheduling trade-offs that may not be valid for a heterogeneous architecture on a single chip. The presented infrastructure allows the evaluation of runtime scheduling techniques for such heterogeneous on-chip architectures, which is not possible using state-of-the-art trace-driven tools, and more difficult for execution-driven approaches (as they may require a full compilation toolchain for the heterogeneous simulated architecture).

Power-aware scheduling for large-scale architectures is another interesting research area, which is explorable using our methodology. Multithreaded applications may trade off performance with power savings for non-critical code regions, resulting in non-significant overall performance degradations. Analyzing those trade-offs is, again, not doable for models on existing trace-driven simulators, and may be hard using measurements in real machines as the access to power sensors is in many cases limited (only a set of sensors are readable at a given point in time) or restricted (only available at the hypervisor level).

Depending on the application, the runtime system interference may be significant. Most applications do not suffer from the runtime system being a bottleneck, so its performance is, in most cases, not critical. However, applications using fine-grain tasks or using fine-grain synchronization semantics may actually see an impact on the overall application performance when runtime system implementations get more efficient. For future architectures with tens or hundreds of cores, the runtime system overhead is likely to be larger, as it may need to handle larger data structures and applications will need to exploit more parallelism, thus going more fine-grain.

In this sense, one of the critical parts of runtime systems is scheduling. There has already been proposals for hardware acceleration of task scheduling for the exploitation of fine-grain parallelism (e.g., Carbon [16]). In this direction, a framework like the presented in Section III not only allows the aforementioned exploration of new architectures or runtime system implementations/optimizations separately, but also hardware-software co-design of runtime systems including hardware support for programming models.

As mentioned in previous sections, the presented methodology also provides dramatic reductions in the tracing time and effort. A single trace represents the intrinsic behavior of the application, and it is not necessary to generate traces for different architecture configurations of the target parallel system. Applications are traced once, and used to evaluate many configurations. Also, there is no need to regenerate traces for following developments in the runtime system. Whenever a new feature or optimization is applied to the runtime system implementation, it is immediately available to the integrated simulator – dynamic component framework, as the runtime system library is used unchanged as the dynamic component.

The time and effort required for tracing applications has been a major issue in research based on trace-driven simulation so far. Thus, by minimizing the required tracing time and effort, new multithreaded applications can be quickly traced and incorporated as evaluation benchmarks to research projects.

V. Related Work

As shown in Figure 1, the vast majority of conference papers that use simulation to evaluate architectural components adopt execution-driven tools. Even though most works use custom performance models, they employ off-the-shelf simulation infrastructures (like Simics [17], SESC [18], Simplescalar [19] or M5[20]) that are modified or extended with those models.

Execution-driven simulators have become more popular compared to trace-driven ones due to the inability of traces to capture the timing-dependent execution interleaving of multithreaded applications. However, execution-driven simulation is time-consuming because it requires the full execution of target applications for its proper operation. A largely-applied solution to this problem is the combination of *sampling* and *checkpointing*. Using sampling, only the most representative execution intervals (and the corresponding warmup phases) are simulated, thus dramatically reducing simulation time. For representative intervals to execute properly, the architectural state of the machine is checkpointed at a specific interval rate. Then, at simulation time, the architectural state is restored to a checkpoint, and simulation starts from there to collect simulation statistics for representative interval executions.

However, checkpointing ends up being equivalent to obtaining and simulating application traces. The runtime and architected state of the application are dumped to the checkpoint, and simulation resumes from that saved state. As such, checkpoints suffer from the same limitations as traces: the saved parallel state of the application does not change for different target architectures or runtime thread managements, loosing the advantage of execution-driven simulation.

Some other simulation frameworks use functional emulators or dynamic binary instrumentation tools that feed the performance models. COTSon [21] uses an AMD functional emulator that receives feedback from the performance simulation engine in order to adjust the execution throughput, and achieve a proper combined timing. Simulators based on dynamic binary instrumentation are used to "trace" the application *online* and generate events that feed models of specific parts of the target architecture (e.g., interconnection network, memory system). These approaches have the same requirements as execution-driven simulation of fully executing the target workloads. However, this kind of simulators are usually fast, because they just emulate or execute natively, but are less flexible for detailed simulation, therefore, being limited to specific evaluations.

Trace-driven simulators are widely used in the industry due to benchmarks and applications not always being available. Such codes (or their input sets) may often be confidential. Also, traces provide a stable reference for the comparison in the evolution of different processor generations. Sampling has also been widely used in trace-driven simulation leading

to very large simulation time savings, with little effort due to the fact that application data is not needed, and the use of techniques like checkpointing is not required. However, the limitations of trace-driven tools on true multithreaded applications that have been discussed, restrict their use to single-core studies or multi-cores running multiprogrammed workloads.

Our simulation methodology positions itself between trace- and execution-driven approaches. Application code independent to the dynamic multithreaded behavior is captured and simulated by employing a trace-driven simulation engine. Then, the parallelism-management operations that depend on the architecture and machine state are dynamically executed by an execution-driven component.

Multithreaded applications can now be simulated using trace-driven environments, thus getting the flexibility of trace-driven simulation (no need for actual application or input data) and the ability of execution-driven approaches to reproduce the dynamic multithread behavior for the target architecture.

VI. Conclusion

The simulation methodology presented in this paper is, to the best of our knowledge, the first hybrid methodology combining traces and execution for parallel application simulation. Traces are generated for the intrinsic application behavior, and the parallelism-management computation is re-executed at simulation time.

One of the contributions of this methodology is a significant reduction in the effort for generating traces. Multithreaded applications are represented by a single trace, that can be used to evaluate any hardware configuration. Also, by using multithreaded applications written in high-level programming models, the instrumentation of the corresponding runtime system library is enough to generate traces for any application.

The use of traces in our methodology provides all the benefits of trace-driven simulation. The most important advantages over execution-driven simulation that we have identified are: the simulation of confidential applications and data, the simulation of applications with very large input data sets (as no data is needed in trace-driven tools), and a larger simulation flexibility (i.e., simulation of very large target architectures, or architectures that do not exist - no compiler is available for them).

Finally, the described methodology provides a wide space of research opportunities. It does not only allow simulation-based research on new large-scale parallel architectures, but also provides an interesting platform for research on runtime systems.

Acknowledgment

This research is supported by the Consolider program (contract No. TIN2007-60625) from the Ministry of Science and Innovation of Spain, the ENCORE project (ICT-FP7-248647), and the European Network of Excellence HIPEAC-2 (ICT-FP7-217068). Felipe Cabarcas is supported in part by Programme Alβan, the European Union Program of High Level Scholarships for Latin America (scholarship No. E05D058240CO). Yoav Etsion is supported by a Juan de la Cierva Fellowship from Ministry of Science and Innovation of Spain. We also want to thank Aamer Jaleel for his valuable comments to improve this paper.

References

[1] M. Pavlovic, Y. Etsion, and A. Ramirez, "Can Manycores Support the Memory Requirements of Scientific Applications?" in *A4MMC'10: 1st Workshop on Applications for Multi and Many Core Processors*, 2010.

[2] R. M. Fujimoto, "Parallel discrete event simulation," in *WSC '89: Proceedings of the 21st conference on Winter simulation*, 1989, pp. 19–28.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, no. 8, pp. 207–216, 1995.

[4] A. Rico, F. Cabarcas, A. Quesada, M. Pavlovic, A. J. Vega, Y. Etsion, and A. Ramirez, "Scalable simulation of decoupled accelerator architectures," Universitat Politècnica de Catalunya, Tech. Rep. UPC-DAC-RR-2010-14, 2010.

[5] "Nanos++ project website," http://nanos.ac.upc.edu/projects/nanox.

[6] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: a Programming Model for the Cell BE Architecture," in *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006, p. 86.

[7] J. M. Perez, R. M. Badia, and J. Labarta, "A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures," in *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, 2008, pp. 142–151.

[8] "Mercurium project website," http://nanos.ac.upc.edu/content/mercurium-compiler.

[9] C. keung Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and R. K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Programming Language Design and Implementation*, 2005, pp. 190–200.

[10] L. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet *et al.*, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Transactions on Mathematical Software (TOMS)*, vol. 28, no. 2, pp. 135–151, 2002.

[11] A. Rico, A. Ramirez, and M. Valero, "Available Task-level Parallelism on the Cell BE," *Scientific Programming*, vol. 17, no. 1-2, pp. 59–76, 2009.

[12] S. Borkar, "Thousand core chips: a technology perspective," in *DAC '07: Proceedings of the 44th annual Design Automation Conference*, 2007, pp. 746–749.

[13] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: programming the memory hierarchy," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006, p. 83.

[14] J. Reinders, *Intel threading building blocks*. O'Reilly, 2007.

[15] 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 *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2005, pp. 519–538.

[16] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: architectural support for fine-grained parallelism on chip multiprocessors," in *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, 2007, pp. 162–173.

[17] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[18] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC simulator," http://sesc.sourceforge.net, 2005.

[19] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[20] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, 2006.

[21] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "COTSon: infrastructure for full system simulation," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 1, pp. 52–61, 2009.