Efficient, Precise-Restartable Program Execution on Future Multicores

Gagan Gupta, Srinath Sridharan, and Gurindar S. Sohi
Department of Computer Sciences, University of Wisconsin-Madison
{gagang, sridhara, sohi}@cs.wisc.edu

1. INTRODUCTION

Multicore processors are becoming ubiquitous, placing new demands on hardware and software designers. No longer do a small set of experts develop a few software applications for a small number of parallel machines. Already standard in servers, desktops and laptops, today handheld devices use multicores, expanding the spectrum of their use from mobile computing at the low end to cloud computing at the high end. Consequently, dramatically increased number of software developers are creating hundreds of thousands of applications to run on a plethora of diverse platforms. Thus ease of writing parallel programs, to achieve energy and/or performance efficiency, continues to gain importance.

At the same time, programmers have to account for the changing characteristics of emerging technologies. Processors are transitioning from homogeneous cores to heterogeneous cores with disparate performance/energy characteristics. As future computing hardware pushes the limits of semiconductor technology, it will become increasingly unreliable. Simultaneously, emerging use of computing systems will require them to host multiple applications concurrently, even on mobile devices. Unreliability, resource (computing and energy) management, and service-level agreements will lead to imprecise knowledge of available resources during a program's execution. Hence programmers can no longer assume availability of given (or constant) resources to process an application, unlike in canonical parallel programming.

The confluence of the above factors pose daunting challenges to programmers in writing ubiquitous programs and achieving their reliable, energy-efficient, parallel execution, while remaining agnostic of the unpredictable, dynamically (and potentially continuously) changing computing conditions.

We propose a model that seamlessly addresses this range of challenges. It relies on expressing parallel algorithms as sequential programs, i.e., *statically-sequential* (§2.1), and performing their controlled, dynamic parallel execution while honoring their sequential semantics. Although at first glance the approach may appear antithetical to parallelism, we show that it affords several advantages. Its intuitive interface and sequentially determinate execution (which ensures that in any execution of a program with the same inputs, a variable is assigned the same sequence of values) allow programmers to easily reason about the program execution, simplifying programming. The model utilizes the implied order in a statically-sequential program to achieve a dataflow schedule of parallel execution (§2.2), potentially exploiting all available parallelism. Further, the order permits the adaptability needed to achieve efficient execution in dynamically changing (§2.3), unreliable (§2.4) computing environments. We provide an overview of these aspects and present results from our efforts to develop several benchmark applications using the model, implemented as a fully functional runtime system, on stock multicore systems.

2. DYNAMIC PARALLELIZATION OF SEQUENTIAL PROGRAMS

Our approach strives to minimize the burden on programmers. It allows programs to be authored in established imperative programming languages, such as C++, and automates their parallel execution. The model extracts a program's computations, establishes the dynamic data-flow between them, and schedules their ordered execution as the prevailing resources permit. It can also roll back the execution, up to a desired point, and resume it, if desired. We highight the model's principles by describing the programming interface and the mechanisms as implemented in the runtime (a C++ library).

2.1 Composing Programs

Programmers today follow modern software engineering and object-oriented (OO) design principles by composing programs from reusable functions that manipulate encapsulated data and communicate with each other using well-defined interfaces. Often such "well-composed" functions avoid side-effects by only manipulating data communicated through the interfaces. We seek to exploit the properties of such OO programs and the natural insights programmers have in their algorithms.

Programs written using the runtime library closely resemble their sequential versions intended to run on a uniprocessor, but for few user-annotations. Users compose programs from computations and data structures amenable to concurrent execution, as they would conventional parallel programs. In addition, they annotate the code to identify *potentially* concurrent functions and the data potentially shared between them. They further formulate the shared data read and written (in the form of objects) by the functions, available from the function's interface, into read and write sets, respectively. Beyond these annotations the onus is not on the user to schedule execution of the computations or to en-

sure independence of concurrent computations, in contrast to conventional parallel programmming.

2.2 Executing Programs

To execute a program on processing cores the runtime raises the granularity of computations to functions. It sequences through the program sequentially but seeks to execute the functions concurrently. Before executing a function the runtime establishes its dependence on already executing functions using the objects in the function's read and write sets. Since objects in the read and write sets may be unknown statically, their identity is established dynamically, at run-time, by dereferencing pointers. The runtime employs dataflow execution since it naturally exposes the innate parallelism between computations. Functions found to be independent are submitted for execution while those that are dependent are "shelved" until their dependences have resolved. The runtime continues to seek work beyond stalled computations, resources permitting, and thus dynamically exploits any available parallelism. Moreover, it ensures that the execution proceeds as per the implied semantics that programmers have come to expect from sequential programs.

The runtime also provisions to handle functions (identified by the user) which do not follow OO principles (e.g., with unknown side effects) by executing them sequentially.

Statically-sequential applications (blackscholes, barneshut, bzip2, dedup, histogram, and reverse index) from standard benchmark suites, developed using the runtime on three stock multicore systems, an 8-thread Intel Nehalem-based machine, a 16-core and a 32-core AMD Opteron-based machines, achieved speedups (harmonic mean) similar to their Pthread versions on the Nehalem machine and over 20% better on the AMD Opteron machines [1].

2.3 Time- and Energy-Efficient Execution

Utilizing resources efficiently in dynamically changing environments will be a key challenge going forward. Doing so will require exposing application parallelism that best fits the capabilities of resources in the execution environment. While exposing too little parallelism can underutilize the resources, exposing excessive parallelism can lead to contention for resources, potentially degrading its time- and energy-efficiency. Dynamically matching the exposed parallelism with the changing capabilities of the execution environment requires the ability to suspend already executing computations, reintroduce them later, and introduce new computations into the environment, as appropriate. The runtime exploits the implied ordering in statically-sequential programs to choose computations judiciously when regulating the parallelism, while ensuring forward progress. It uses a *Goodness of Parallelism (GoP)* metric, computed periodically as the execution unfolds, to correlate the instantaneous efficiency of the program to the instantaneous degree of parallelism. A drop in efficiency causes it to throttle the parallelism to ease contention, while an improvement in efficiency causes it to increase the parallelism to exploit available resources.

Experimental results on a stock 4-core (8-thread) Intel Core i7 2600 (Sandy Bridge) workstation show that our approach achieves up to 50% higher time- and energy-efficiency over the state-of-the-art parallel execution systems across a variety of dynamic operating conditions.

2.4 Precise-Restartable Execution

Future computer systems will present unreliable resources to applications due to exception events, e.g., hardware faults, timing errors caused by aggressive energy management, or due to resource management. To be efficient it will still be desirable to continue executing the interrupted program, possibly at a different time and/or on another system, without discarding all of the completed work. Hence to resume execution in such scenarios the runtime supports *precise-restartability* of parallel programs, analogous to precise-interruptible execution of sequential programs.

The runtime exploits the implied ordering to precisely identify the excepted computation in the statically-sequential program and restores the program state to reflect the sequential execution of the program up to the computation. To do so it tracks the invocation and completion of computation in the implied program order. Further, it checkpoints the state a computation may modify, i.e., its *mod set* (a user-provided set similar to the computation's write set and processed similarly) before its execution. Once the excepting condition is mitigated the program may resume from the excepting computation. The runtime also incrementally checkpoints the program state after each computation successfully completes, using its mod set. This state can be used to spatially or temporally migrate a halted program.

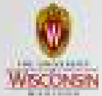
Experiments on a stock 12-core (24-thread) Intel Xeon E5-2420 (Sandy Bridge) workstation show that the runtime can tolerate signficantly higher (proportional to thread-count) exceptions than the conventional approaches. Depending on the application, the support to tolerate aggressive exception rates (e.g., up to 2 every second) incurs performance overheads ranging from 0% to 135% (at 0 faults).

3. CONCLUSION

Parallel programming for multicore-based systems and their dynamically changing operating environments pose significant challenges to everyday programmers in the effort to improve productivity and to achieve error-free, efficient execution of their programs. We presented a model that meets these challenges better than other approaches by using statically-sequential programs and performing their dynamically controlled dataflow execution.

References

[1] G. Gupta and G. S. Sohi. Dataflow execution of sequential imperative programs on multicore architectures. In *MICRO-44*, December 2011.



Efficient, Precise-Restartable Program Execution on Future Multicores

Gagan Gupta, Srinath Sridharan, and Gurindar S. Sohi

Challenges in Future Computing Systems

- **Programmer Productivity**
  - Simplify programming of dynamic and static heterogeneous multicores
  - Enable portable, platform-agnostic programming
- **Efficiency**
  - Optimize energy- and performance-efficiency
  - Adapt to dynamically and continuously changing operating conditions
- **Reliability**
  - Tolerate increasingly unreliable resources
  - Provide differentiated levels of service

Proposed Model: Sequential Programs, Dynamic Parallelization

- **Statically-sequential** programs (using parallel algorithms)
- **Dynamic dataflow** parallel execution
  - Preserving sequential semantics
- **Dynamically controlled** parallelism
- Implemented with a software runtime library (C++)
  - Seamlessly addresses Productivity, Parallel Execution, Efficiency, and Reliability

Programming

- Leverage modern **Object-Oriented** principles
  - Modularity, encapsulation
- Exploit **users' insights** in their algorithms
  - Users identify *potentially* parallel functions, data *potentially* shared between them, and their read and write sets
- Users do not ensure independence between computations, nor orchestrate parallel execution

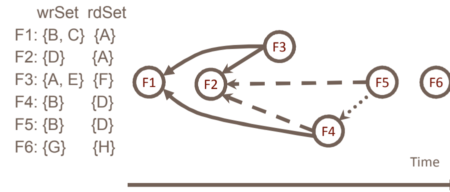
```
1   for (i = 1; i < 7; i++) {
2     df_execute (&F, wrSet, rdSet);
3   }
```

Example of Statically-Sequential Code

- **Simplified parallel programming**

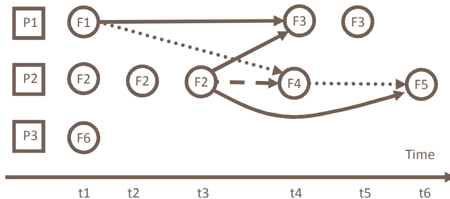
Parallelizing the Execution

- Exploit **Function-Level parallelism**
- Program execution unfolds sequentially
  - Functions execute concurrently (dataflow schedule)
- Data dependences between functions established dynamically (using data sets)

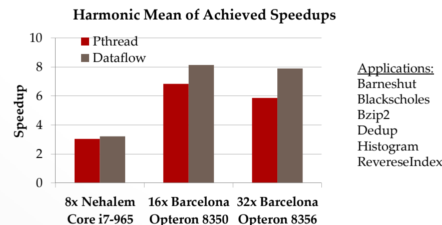


Dynamic Invocations of Example Code and Dependence Graph

- Independent functions execute concurrently, dependent functions are serialized (in program order)
- Dependences are tracked as functions execute and complete



Example Dataflow Execution Schedule on 3 Cores



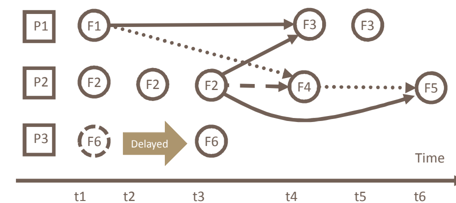
- **Up to 20% higher speedups than conventional implementations**

Benefits of Order and Dataflow Execution

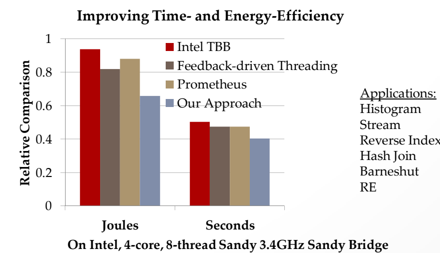
- Dynamically establish dependences to honor
- **Sequentially determinate**, predictable and repeatable execution
- Freedom from deadlocks; guaranteed forward progress
- **Arbitrary control of execution** to optimize efficiency
- **Precise-restartability** of halted computations

Efficient Execution

- "**Goodness of Parallelism**" metric to assess instantaneous efficiency
  - Measured periodically
- **Adapt degree-of-parallelism** to resource contention
  - Optimize for time- and energy-efficiency



Dynamically Adaptive Execution: F6 Delayed to Improve Efficiency



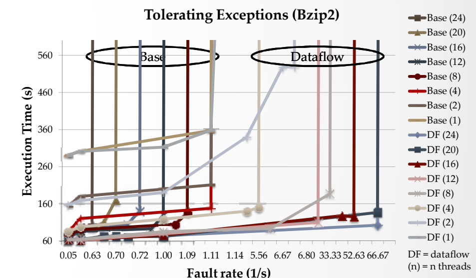
- **Up to 50% more time- and energy-efficient than state-of-the-art parallel execution systems**

Precise-Restartable Execution

- Order of executing computations tracked using a **Reorder List**
- Functions "retired" in program order
- Computation state checkpointed in **History Buffer**
  - Restored on exception, if needed

Epoch	Reorder List Entries	Completed	Retired
t1	F1 F2 F3 F4 F5 F6		
t2	F2 F3 F4 F5 F6	F1	F1
t3	F2 F3 F4 F5 F6	F1, F6	
t4	F3 F4 F5 F6	F1, F6, F2	F2

Precise-Restartable Execution



- **Tolerates significantly higher fault rates (proportional to thread-count) than conventional methods**
- **Incurs 0% to 135% performance overhead (at 0 faults)**

Program Execution on Future Multicores

Dynamically controlled, dataflow execution of statically-sequential programs

- **Enables simplified, platform-independent programming**
- **Automates platform-specific, dynamic and continuous optimizations for energy- and performance -efficiency**
- **Tolerates high fault rates and manages resources at low overheads**