Program Execution on Multicore and Heterogeneous Systems

Gagan Gupta, Srinath Sridharan, and Gurindar S. Sohi

Abstract

As uniprocessor performance fails to improve at the erstwhile rates, computer architects are turning to multicore architectures to stay on the performance curve. Parallel programming holds the key to unlocking the performance benefits of such architectures. Conventional parallel programming models require programmers to abandon the well established sequential programming paradigm, and write parallel programs. Such models are fraught with pitfalls such as deadlocks, livelocks, nondeterminism, etc. Recently Allen et. al. [3] introduced a new dynamic dependence based programming model, called Serialization Sets, which maintains the sequential program semantics and yet delivers performance comparable to the conventional methods. While we extended the Serialization Sets model in [21], to allow programmers to express data dependencies between computations, and extracted additional parallelism, the model still had limited capability to discover parallelism. In this work, we propose a generic dataflow execution model capable of exploiting all possible parallelism, using the same imperative programming framework. We present the details of the implementation and show that the new model improves the performance of the applications we developed by 15%-25% on a general purpose multicore architecture, and at the same time scales well with the increase in the number of processors. We also apply this framework to another class of architecture, GPGPU, and show we are able to derive similar performance benefits while greatly simplifying programming for such architectures.

1 Introduction

The computer industry has transitioned from an era where microprocessor clock speeds doubled every two years, automatically delivering software performance improvement, to an era where microprocessor speeds are improving only incrementally. Computer designers have turned to integrating multiple cores on a processor, offering parallel execution as a new avenue for software performance improvement. On one hand vendors have introduced general purpose multicores [13] based on the familiar single-core architectures, on the other they have also introduced GPGPUs [17] which consist of a large number of cores and are ideally suited for data parallel algorithms [12]. Parallel programming plays a critical role in exploiting the performance benefits of such processors, and the best way to do so remains an open question. While general purpose processors are relatively easier to program, GPGPUs and their prevalent programming framework, CUDA, pose a significant challenge to programmers. Today, the prevalent model for programming both of these architectures is multithreading. But multithreading has significant drawbacks that threaten to hamper its widespread adoption [15]. Dependencies in a multithreaded program are typically encoded statically, making it hard to apply the model to a wide range of applications with irregular data accesses that will now need to achieve parallel execution. Multithreading also introduces many new types of errors, such as data races and deadlocks, not present in sequential programs. Furthermore, multithreaded programs execute nondeterministically, making it very difficult and costly to develop, test and debug these programs. It may also result in significant degradation in application reliability.

Recently, Allen et al. [3] have proposed a novel parallel execution model named, Serialization Sets. The model was further enhanced by us in [21]. Serialization Sets provide a programming abstraction that retains much of the simplicity of the sequential model and conveys dependence information to a runtime system that performs opportunistic parallelization of independent computations. Initial evaluations in [3] and [21] show that Serialization Sets can achieve parallelism comparable to multithreaded execution, but with a sequential program representation that lends to a determinate, race-free, parallel execution.

Serialization Sets strive to ease the programming effort by retaining the sequential interface and automatically handling the write-after-write and read-after-write dependencies between computations. However, the model misses the opportunity to fully exploit parallelism inherent in the code. In this work we incorporate an execution model in the runtime system that dynamically discovers the dataflow within the program and exploits it to achieve maximum parallelism. Given this generic dataflow model and a sequential programming interface, we also extend it to provide a programming framework for

GPGPUs which have an architecture very different from the general purpose architectures. In the remainder of this paper we present the details of our implementation and the results from the effort to develop a few applications on the respective platforms.

The rest of the paper is organized as follows. In section 2 we provide an overview of Serialization Sets and discuss the motivation for this work. Section 3 describes the implementation of the new execution model. Section 4 describes the applications and their implementation using the new model. In section 5 we extend the model to GPGPUs. In section 6 we briefly describe some additional work performed but not yet fully evaluated. In section 7 we provide a brief overview of related work. Section 8 discusses potential future work and concludes.

2 Serialization Sets

Multithreaded programming model requires programmers to statically encode independence among computations in the program. They are also required to synchronize accesses to shared data from these mostly independent computations. Serialization Sets take a radically different approach [3]. They rely on sequential program representation, but take a data-centric view to parallelization. Currently Serialization Sets support the C++ language, which has been extended using STL. Programmers use encapsulation to identify data and computations that operate on them. Programmers encode which other objects these computations may be dependent on. They further identify computations that can potentially be performed in parallel. They also associate data objects, on which dependencies can occur, with "serializers". Serializers map operations on the same data to the same "serialization set", and computations on different data to different serialization sets. The model includes a runtime scheduler that dynamically discovers and exploits parallelism as the execution unfolds. In the process, it may execute operations in different serialization sets on different processors. However, computations from the same serialization set are executed sequentially on the same hardware context (referred to as threads hereafter) to honor the data dependencies. (The model also provides an interface to programmers which can be used to ensure all computations in a serialization set have completed.) Thus Serialization Sets strive to incorporate dataflow execution using an imperative language such as C++. Programmers can write most parallel programs using this basic infrastructure. The few examples evaluated in [3] and [21] show Serialization Sets can achieve performance similar to their respective pthread implementations.

Although the runtime scheduler attempts to perform dataflow execution, it blocks resources when it encounters

```
1    ..
2    begin_nest
3      A.delegate(method1);
4      A.delegate(method2);
5      B.delegate(method3);
6      ..
7      C.delegate(A, B, method4);
8      D.delegate(method5);
9      ..
10     ..
11   end_nest
14   ..
```

Figure 1: Example of Serialization Set-based code.

computations that are dependent on other data currently being computed. While this is required to resolve dependencies and ensure correct execution, blocking resources limits the scheduler's ability to discover and exploit additional parallelism while computations wait for their dependencies to resolve. Consider the Serialization Sets pseudocode shown in Figure 1. Line 3(5) signifies that the object $A(B)$ has been associated with a serializer (when $A(B)$ was declared) and method *1*(*3*) on $A(B)$ may be *delegated* for execution in parallel. Both, methods *1* and *2* operate on object *A*. The runtime scheduler serializes execution of method *2* behind method *1* if method *1* is in flight when method *2* is encountered, thus automatically resolving the "write-after-write" dependence. Line 7 signifies that the method *4* on object *C* reads data from objects *A* and *B*, and may be *delegated* for parallel execution. Thus a "read-after-write" dependence exists between method *4* and methods *2* and *3* (on lines 4 and 5). For correct execution, method *4* cannot proceed until methods *2* and *3* finish. The current Serialization Set model blocks a resource while method *4* waits for methods *2* and *3* to complete. This limitation restricts discovery of potential parallelism in the downstream code. For example, in Figure 1, method *5* on line 8 could have executed in parallel with methods *2* and *3*, but cannot since method *4* has blocked a computing resource. Thus we conclude that there is an opportunity to achieve higher performance than the model currently does.

We support our intuition by inspecting two real life applications: network processing and data compression. Figure 2 shows the main processing loop of the networking code (described in further details in section 4.1). The *Update* method on line 7(13) can block a resource while waiting for *Classify* methods from lines 3 and 4(9 and 10) to complete while independent methods in the code downstream (lines 9, 10 and 3, 4 of next iteration) could have been executed. In this example,

2 out of possible 6 threads get blocked and there is an opportunity to gain about 50% more performance.

```
1   for (all packets) {
2
3     pkt1.delegate(Classify, HRC1);
4     pkt2.delegate(Classify, HRC1);
5
6
7     HRC1.delegate(pkt1, pkt2, Update);
8
9     pkt3.delegate(Classify, HRC2);
10    pkt4.delegate(Classify, HRC2);
11
12
13    HRC2.delegate(pkt3, pkt4, Update);
14  }
```

Figure 2: Pseudocode for Packet Classification.

In another example, Figure 3 shows the main processing loop of the bzip2 code. The *opfile* method on line 5 blocks a resource while it waits for the *compress* method of the previous iteration to complete, whereas the *compress* method of next iteration could have utilized the resource. In this example, while it may seem there is an opportunity to almost double the performance, it would be unlikely since file I/O is part of the algorithm and that step is sequential.

```
1   prev_block = NULL;
2   for (all blocks) {
2     ..
3     block = new block_t();
4     block->delegate(compress);
5     block->delegate(prev_block, opfile)
6     prev_block = block;
7     ..
8   }
```

Figure 3: Pseudocode of Serialization Set-based bzip2.

The contributions in this work are two-fold. First, we augment the runtime system to make it non-blocking and support the execution of dependent methods in a truly dataflow fashion, thus allowing for more opportunistic exploration of parallelism. Second, we extend this model, especially the sequential programming framework, to GPGPUs. To the best of our knowledge this is the first effort to apply sequential programming model to exploit parallelism using GPGPUs. We also evaluate the impact on performance of the new models.

3 Implementation

Using the C++ STL interface in Serialization Sets, programmers code objects to be associated with serializers. Serializers provide an interface to the runtime scheduler, and execution of methods on the objects is coordinated using them. The runtime uses serializers to track "write-after-write" dependencies. When one occurs, methods are "queued" in the serializer for sequential execution which avoids data races. This interface also is extended to support "read-after-write" dependencies [21]. For example, in Figure 1 the statement *C.delegate(A, B, method4);* indicates to the runtime system that *C* "may be" dependent on *A* and *B*. The runtime system ensures that method *4* operating on *C* does not execute until both *2* and *3* complete execution. As stated earlier, however, method *4* occupies a thread until the condition is not satisfied.

3.1 Achieving Dataflow Execution

Figure 4 depicts the mechanism used in the runtime system to support the execution of dependent methods in a dataflow fashion without blocking resources. The presence of a call to *delegate* in the program causes the compiler to instantiate an invocation object for the specified method call. The invocation object contains a pointer to the object and the method to be delegated, as well as the specified arguments. It also contains the serialization set identifier that allows the runtime to detect erroneous serializers. The runtime system processes the code sequentially. When it encounters a *delegate* directive, it executes the serializer to compute the serialization set identifier for the method invocation. It then queues the method invocation in the corresponding serialization set.

The runtime scheduler maintains a task queue for each thread in the system. It uses task stealing to dynamically distribute and balance the load across the resources in the system. As the main context of the program executes, the runtime discovers delegated methods, resolves dependencies and attempts to execute them in parallel. When a thread discovers that a method is ready for execution (there are no dependencies to wait for) it pushes the program continuation (part of the program after the method call) on to its task queue and begins execution of the method. Upon completion of the method, the thread looks for the next quantum of work in its task queue, where it may find the continuation that was pushed earlier. Alternatively, another thread may have "stolen" the continuation when it was looking for work. (This is a provably optimum scheduling policy [8].) Thus work is dynamically discovered, distributed and balanced across the computing resources in the system.

We illustrate the execution flow in Figure 4 using the code in Figure 1. We pick up execution from line 4. Figure 4(a) depicts the state of the runtime system after delegating methods on line 3 and 4 in the code. Methods *2* and *3* are launched for parallel execution, but in the sequential program order. Serializers S_a

and S_b (not shown in the figure) corresponding to the methods are scheduled in the runtime system and their corresponding invocations I_a and I_b are queued in the respective serialization sets. While these invocations are ready to get executed, the runtime system proceeds and launches method *3* in program order. Serializer S_c corresponding to *C* is scheduled and the method invocation I_c is queued in its set. The runtime system interprets I_c to be a special invocation (as specified by the programmer) which has data-dependence on the previously launched invocations I_a and I_b. It therefore generates two tokens W_1 and W_2 and queues them behind I_a and I_b in the S_a and S_b serialization sets, respectively. Tokens are nothing but place holders and I_c will not execute until all the tokens that it generated are returned back. This process is depicted in Figure 4(c).

In Serialization Sets, a spinlock solution is employed for the invocations that are data-dependent on preceding invocations. For example, I_c spins until it gets back all its tokens, thus blocking a resource. We overcome this limitation as follows. A scheduled method is immediately "shelved" if it needs to wait for its tokens. After shelving the method, the thread is free to look for other work in the program context that may be ready for execution. For e.g., in Figure 4(d) method *4* is shelved and the runtime schedules method *5* for execution. Serializer S_d corresponding to *D* is scheduled and the method invocation I_d is queued in its set.

Shelved methods are essentially waiting for input tokens to arrive. The thread that returns the last token to the method is responsible to detect the case and enqueues the serializer in its work queue. The serializer will be scheduled for execution the next time the thread looks for work in its work queue, and the ready method will be executed. Adding this capability required us to make the work queue polymorphic. While it held only continuations before, it now also needs to hold serializers. The two types of work quanta are handled differently. Continuations require establishment of the continuation's native stack before it can be executed. If another delegated method is encountered, the continuation from thereon is pushed onto the work queue and the method is executed. Executing serializers, on the other hand, is similar to invoking methods and can be executed using the scheduler's stack. It is also possible that a method in the serializer is not ready for execution and the serializer is once again shelved. Note that when a serializer is shelved, it is not enqueued in any work queue. Only when the first method in the serializer receives its last token does it get enqueued in a work queue. Hence a thread has to detect that it has returned the token to the first method in the serializer.

Now that it can hold methods not ready for execution, the serializer interface had to be modified. Before, once an invocation is dequeued from the serializer, it was either executed or the thread would poll until all the tokens arrived. With the new capability, if a method is not ready the serializer has to be shelved. Hence, now methods are dequeued from the serializer only after the thread ensures that it is ready for execution.

Adding above functionality required us to handle the following race conditions. A thread may be returning a token to the dependent method while the thread on which the serializer is scheduled may be checking whether the method is ready for execution. A more critical race condition is when a thread is returning the last token. The returning thread may try to enqueue the serializer in its work queue and the thread on which the serializer is already scheduled may attempt to execute the method. Appropriate counter-based synchronization mechanism is utilized to handle these race conditions.

Returning back to our example, as soon as I_a completes execution, S_a returns the token W_1 back to I_c as shown in Figures 4(e) and 4(f). I_c still cannot execute as it has not received token W_2 yet. In the next time step, I_b completes execution and S_b returns the token W_2 back to I_c (Figure 4(g)). Once I_c has received all its tokens, it can be scheduled for execution as shown in Figure 4(h). When I_c completes, S_c can be descheduled from the runtime system.

4 Evaluation

To evaluate the implementation of the new model we used programs from two diverse application domains. The first is a networking application that classifies packets. The second is the commonly used data compression program, bzip. In this section we use small code snippets to highlight how they benefit from the new dataflow execution model.

4.1 Packet Processing using Hyper Rule Cache

Recently, Dong et al. have proposed an advanced version of the packet classification algorithm [7], Hyper Rule Cache, amenable to implementation in software. Due to ample parallelism inherent in packet processing, the algorithm lends itself well to multicore architectures. The first step in packet processing is to classify a packet and determine how it will be processed. The header from the packet is used to lookup a rule table which returns the associated action. Since lookup is typically performed using CAMs, external to the packet processors, this step consumes a large number of cycles. In the new scheme the authors propose an algorithm that maintains a small subset of rules in a software-managed rule cache, kept local to the packet processors. Attempt is made to classify packets by first looking up the local rule cache, failing which the entire rule-set is used. The local rule cache is periodically

updated with rules needed to classify packets that miss in the cache. In order to not stall the process, two copies of the rule cache are used. One is used to perform the look up while the other is being updated. Copies are switched after each update.

Figure 2 shows the pseudocode of the core operations used to classify packets. In lines 3 and 4, the first set of packets are classified using the first rule cache (HRC1), after which HRC1 is updated (line 7). While HRC1 is being updated the next set of packets are classified (lines 8, 9) using the second rule cache (HRC2), after which HRC2 is updated (line 13). When the programmer identifies input data to a method, as is done in lines 3 and 4 (HRC1 is input), the scheduler automatically ensures that any methods updating the inputs finish before the method begins execution, thus resolving the read-after-write dependence. We use the mechanism for a slightly different purpose in line 7. By specifying *pkt1* and *pkt2* as inputs (even though the *Update* method may not read data from the objects), we cause the system to ensure methods updating *pkt1* and *pkt2* finish before executing the method *Update* on HRC1.

Figure 5 shows the execution time of the packet classification algorithm for implementations using Serialization Sets and the proposed dataflow model. The new dataflow model reduces the execution time by up to 25%, primarily due to its ability to expose a higher degree of parallelism.

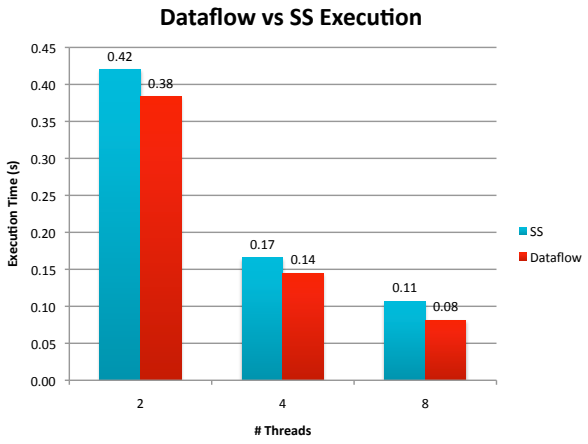


Figure 5: Execution time on 2 socket (8 threads) haled machine.

4.2 Data Compression

Bzip2 is a lossless, block-sorting compression algorithm [9]. It uses the Burrows-Wheeler Transform (BWT) to compress blocks of data from a file and writes the compressed blocks to the disk. Figure 3 shows the simplified version of the bzip2 code written using Serialization Sets. The invariant that is preserved in the algorithm is that the compressed blocks are written in the same order as the uncompressed blocks in the input file. The programmer ensures that before writing a compressed block to the disk, all previous blocks have already been written. Figure 3 shows how this dependence is expressed in the program.

Figure 6 compares the execution times of both Serialization Sets and the new dataflow model for medium and large input set on a 4 socket (8 threads) Nehalem machine. The dataflow model outperforms SS by up to 18%. While we expected a larger improvement in performance, Amdahl's Law strikes in the form of serial file I/O which is included in the execution time.

Performance data for both applications show that the gains increase as the number of threads increase in the system. As expected, increased number of threads make more resources available for the runtime scheduler to discover and exploit parallelism.

5 A sequential programming model for GPGPUs

Modern processor architectures are evolving towards more software-exposed parallelism through two features: multiple cores and wider SI(P)MD accelerators. At the same time, graphics processing units (GPUs) are gradually adding more general purpose programming features. The key software development challenges and complexity that arise from these trends are as follows. First, how to mitigate the software development complexity which comes with the programmer explicitly identifying and representing parallelism? Second, How to provide flexibility to the programmers in order to write code for multiple architectures?

Modern GPU programming models are constrained in such a way that the compiler and runtime cannot reason about the application encoded with a sequential representation and extract the parallelism automatically. They expect the programmer to "think-parallel" while the human brain is tuned to think sequentially. Examples of this include DirectX, CUDA [20], and Cg. If the programmer can reformulate the application to work under GPU constraints, the compiler/runtime can do the rest automatically. However, reformulating the application to fit these constraints often requires considerable programmer effort, and can result in significantly less efficient software algorithms. For example, it is difficult to operate efficiently on linked lists or compressed data structures, so applications that would naturally like to use these types of algorithms must be reformulated to use algorithms more consistent with GPGPU models.

In this project, we propose a new sequential programming model for GPGPUs, *SS-GPU*. To the best of our knowledge this is the first effort to apply sequential programming model to exploit parallelism using GPGPUs. The programmer writes a sequential program with operations specified on individual data objects. The runtime system executes the program in two phases, *Aggregation* and *Execution*.

1) During the *aggregation* phase, the runtime dynamically identifies and groups data objects with same operations into a container. It also dynamically constructs a dependence graph between operations on same data elements based on the order in which the operations are delegated in the sequential program.

2) In the *execution* phase, the operations and their corresponding data elements are launched on to the GPU for execution. The order in which the operations are launched is provided by the dynamic dependence graph constructed in the *aggregation* phase. The runtime system automatically manages transfer and co-ordination of data objects between the CPU and GPU.

The philosophy of SS-GPU is to extract parallelism from sequential program representation. Rather than thinking about threads, cores, vector ISAs and SIMD operations, SS-GPU requires the programmer to encapsulate data into classes and structure programs hierarchically. This thought process is consistent with the principles of object-oriented programming. In our experiments, we ported all the benchmarks to fit the idiomatic object-oriented C++ paradigm.

5.1 Example

Figure 7 depicts an example code written using C++. The code defines a simple class named *example_t* with two *public* methods, *foo* and *bar*. Both the methods operate on the private data members defined in the class (not shown). It then instantiates an stl_vector of objects of type *example_t* of size MAX_SIZE. Within a simple for_loop, the methods *foo* and *bar* are then called on every object in the stl_vector.

Figure 8 depicts the mechanics of how the runtime system executes this program. As shown in Figure 8(a), assume that *foo* and *bar* are the only methods that exist in the system. They both hold pointers to a private container, into which data objects are collected dynamically. As the runtime walks through the loop, it encounters methods operated on different data objects. As and when data objects are encountered, the runtime steers them into their corresponding containers. Figure 8(b) shows of the state of the system after encountering the methods *foo* and *bar* on *object 1*. As soon as the reference of *object 1* is pushed into the container of *bar*, the runtime identifies that *bar* has a data dependence over *foo* and that *bar* has to follow *foo* according to sequential semantics and cannot execute concurrently with foo. Figure 8(c) shows the state of the system after aggregating all the objects into their respective containers. This is a simple example where there is a complete dependence between two methods invocations. That is, all the objects operated on by *foo* is also operated on by *bar* and hence *bar* follows *foo*. There are cases in which there is only partial dependence. In those cases, we need better mechanisms to reduce the container dependence into fine-grained dependence (at object level) or divide the container into multiple independent sub-containers. Solutions to partial dependence are beyond the scope of this project.

Once the dependence graph is constructed, the operations and their corresponding containers are launched onto GPUs. In this example, *bar* is launched after *foo*. The runtime manages transfer of data between CPU and GPU. It also manages mapping of container chunks to one or more GPU cores.

```
Class example_t
{
public:
        void foo();
        void bar();
};
//Main
#define MAX_SIZE 3
Vector<example_t> objects(MAX_SIZE);
for (i=0;i<MAX_SIZE;i++)
{

   objects[i].foo();
   objects[i].bar();
}
```

Figure 7: Example code written using C++.

5.2 Implementation

In this section, we describe the API and design of the C++ library that implements the SS-GPU abstraction through compile time template instantiation and a runtime support library. Programmer uses C++ objects to encapsulate data into disjoint domains. Method calls serve as the granularity of operation that may be delegated, and thus potentially executed in parallel. We implemented the library on top CUDA, a well known programming model for GPUs.

CUDA programming model has many restrictions. One of the biggest challenges is that CUDA does not support runtime polymorphism. It means that indirect method calls on data objects are not allowed. A workaround to achieve the effect of runtime polymorphism is to define what is called as C++ function

object or *functors*. The programmer defines a *functor* for every method in a class that is needed to be executed on GPUs. The functor definition is hidden from the programmer. All that she needs to do is instantiate a functor for every method defined in the class that is needed to be executed on GPU. For example, the functor definitions for the example shown in Figure 7 is shown in Figure 9.

Once the *functors* are defined, the programmer calls the functors on objects using a special *delegate* interface which marks the method for GPU execution. The SS-GPU equivalent of the C++ code in Figure 7 is shown in Figure 10.

```
Class example_t
{
public:
          void foo();
          void bar();
};

Functor_t<example_t> F_foo;

Functor_t<example_t> F_bar;
```

Figure 9: Example code using SS-GPU.

```
//Main
#define MAX_SIZE 3
Vector<example_t> objects(MAX_SIZE);
for (i=0;i<MAX_SIZE;i++)
{

   delegate (objects[i],F_foo);
   delegate (objects[i],F_bar);
}
```

Figure 10: Example code written using SS-GPU.

5.3 Results

We used two programs to evaluate our implementation, saxpy and blackscholes. We ported these benchmarks to our library by first rewriting them as idiomatic object-oriented C++ programs, using standard template library (STL) data structures. We then augmented them with the annotations specified in Section 5.2. All the programs were compiled using the NVIDIA *nvcc* compiler. The baseline is the single threaded execution on Intel 4-core Nehalem machine running at 3.2GHz. The GPU chip used for our experiments is the NVIDIA GeForce 8600 GT. It has 32 stream processing cores running at 540 MHz. The speedup numbers are reported in the Figure 11. The initial speedup numbers are overwhelming and at least 2 times faster than the multithreaded execution on the baseline Nehalem machine (results not reported).

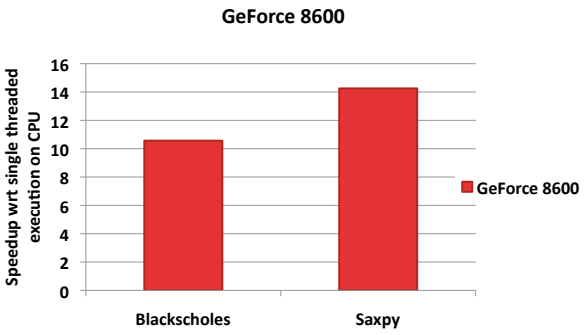


Figure 11: Speedup of SS-GPU

6 Additional Work

The current programming model requires programmers to define methods that operate on single objects. While many algorithms can be coded adopting this philosophy, some, such as database transactions, can not. Database transactions can have read and write sets consisting of multiple objects and require support for atomic execution at the transaction level. Using the underlying runtime system described in section 3, we have created another execution model capable of supporting database transactions and methods that update multiple objects. We have extended the token mechanism and shelving support to achieve this capability. Although we have a functionally working system, we were unable to evaluate the system in time for this report.

7 Related Work

Pthread [1] and OpenMP [6] have long been used to code multithreaded programs. More recent research has resulted in other imperative frameworks such as TBB [18], Cilk [8], Cilk++ [16]. They all provide a sequential programming interface by extending languages such as C, FORTRAN and C++. Such models provide mechanisms to explicitly represent parallelism, along with synchronization primitives. Some of them also provide high level constructs such as parallel *FOR* loops and reduction operators. While these models ease the task for the programmer somewhat, they do not resolve the core issues of deadlock and nondeterminism.

Models such as actors [11], active objects [14], MultiLisp [10], TAM [5] and Jade [19] have similarities with Serialization Sets. Actors and active objects also avoid data races, but are restrictive and use asynchronous communication leading to nondeterministic execution. MultiLisp can execute expressions concurrently with the program, but does not provide support for automatic dependence resolution. TAM employs a dataflow execution approach similar to our proposal. However it requires programmers to use a declarative programming language such as Id, whereas our proposal is based on the more widely used imperative languages such as C++. The Jade language supports deterministic execution and uses access specifiers to identify data read and written by parallel tasks. Access specifiers determine when the data is ready for use by tasks, whereas Serialization Sets send the operation to the owner of the data, and the methods that produce the inputs notify the owner when completed. Deterministic Parallel Java (DPJ) [4] is yet another deterministic model similar to the Jade language. Programmers statically identify read and write sets of methods. However, in DPJ only static analysis is performed, no attempt is made to dynamically exploit parallelism.

CUDA [20] is the prevalent programming framework for GPGPUs. It relies on multithreading and requires explicit management of hardware resources. The proposed SS-GPU model attempts to abstract away the hardware and provides a sequential programming interface.

In another strand of recent research, transactional memories (TM) [2], in the form of hardware, software or hybrid implementations, have been proposed. They provide automatic and optimistic execution of critical sections by dynamically detecting and rolling back the results in case of conflicts. In our proposed dataflow model dependencies are resolved before executing methods and hence conflicts are avoided altogether. While TM provide a much simplified programming interface, programmers have to still identify the critical sections.

8 Future Work and Conclusion

As computer architects explore multicore architectures as an avenue to improve processor performance, parallel programming will play a vital role in their successful adoption. Serialization Sets is a new programming model that strives to retain the well understood sequential programming interface, supports determinate execution and thus eases parallel program development. In this work we have proposed a generic dataflow execution model using an imperative programming framework. While the community is exploring different multicore architectures, we have shown that our programming model is applicable to general purpose as well as GPGPU architectures. We have described the implementation of the new model, and demonstrated that it makes it easy and intuitive to code programs. The new dataflow model achieves 15 to 25% better performance than Serialization Sets on general purpose architecture. It achieves 14x speedup on GPGPUs as compared to single thread implementation in the examples we coded.

Next, we plan to evaluate the new transaction-capable dataflow model. We have only scratched the surface as far as programming GPGPUs are concerned. Another worthwhile effort will be to extend the idea to ease the programmer's task to exploit heterogeneous systems, such as one consisting of general purpose multicores alongside of GPGPUs. Finally, other applications need to be coded using the model to better understand its viability and limitations.

9 Acknowledgment

We would like to thank Matthew Allen for his help in understanding the Serialization Sets runtime system, and other discussions in general.

References

[1] Posix threads programming. In *https://computing.llnl.gov/tutorials/pthreads/*, Livermore, California, USA. IEEE.

[2] A.-R. Adl-Tabatabai, C. Kozyrakis, and B. Saha. Unlocking concurrency. In *ACM Queue*, pages 24–33, USA, December 2006. ACM.

[3] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization sets: a dynamic dependence-based parallel execution model. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 85–96, New York, NY, USA, 2009. ACM.

[4] R. L. Bocchino Jr. and et al. A type and effect system for deterministic parallel java. In *OOPSLA: Object Oriented Programming, Systems, Languages and Applications*, USA, October 2009. ACM.

[5] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. Tam—a compiler controlled threaded abstract machine. *J. Parallel Distrib. Comput.*, 18(3):347–370, 1993.

[6] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. In *IEEE Computation Science and Engineering*, pages 46–55, USA, 1998. IEEE.

[7] Q. Dong, S. Banerjee, J. Wang, and D. Agrawal. Wire speed packet classification without tcams: a few more registers (and a bit of logic) are enough. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 253–264, New York, NY, USA, 2007. ACM.

[8] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In

Proceedings of the 1998 conference on Programming Language Design and Implementation (PLDI), pages 212–223, 1998.

[9] J. Gilchrist. Parallel data compression with bzip2. *www.bzip.org*.

[10] R. H. Halstead. Multilisp: A language for concurrent symbolic computation. In *ACM Transactions on Programming Languages and Systems*, pages 7(4):501–538, USA, 1985. ACM.

[11] C. Hewitt. Viewing control structures as patterns of passing messages. In *Journal of Artificil Intelligence*, pages 8:323–363, USA, 1977.

[12] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[13] S. Kottapalli and J. Baxter. Nehalem-ex cpu architecture. In *HOT CHIPS 21*, USA, 2009. ACM.

[14] R. G. Lavender and D. Schmidt. Active object: An object behavioral pattern for concurrent programming. In *PLOP: Proceedings of the 2nd conference on Pattern Languages of Programs*, USA, 1995.

[15] E. A. Lee. The problem with threads. In *IEEE Computer*, pages 33–42, USA, 2006. IEEE Computer Society.

[16] C. E. Leiserson. *Cilk++: Multicore-enabling legacy C++ code*. Carnegie Mellon University Parallel Thinking Series, USA, April 2008.

[17] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28:39–55, 2008.

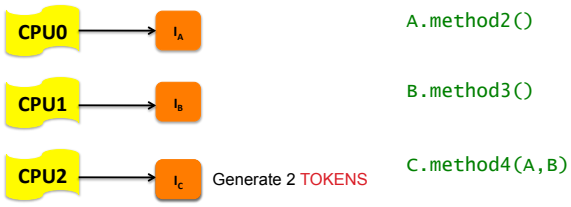
[18] J. Reinders. *Intel Threading Building Blocks*. O'Reilly Media, Inc., 2007.

[19] M. Rinard and M. S. Lam. The design, implementation, and evaluation of jade. In *ACM Transactions on Programming Languages and Systems*, pages 20(3):483–545, USA, 1998. ACM.

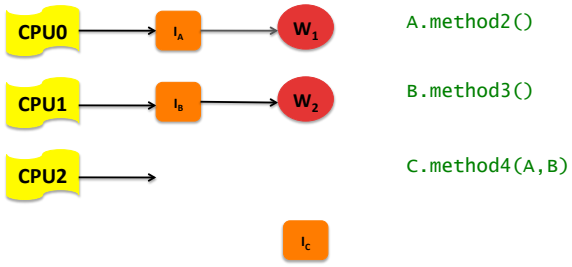
[20] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.

[21] S. Sridharan and G. Gupta. Under, over and beyond serialization sets! In *CS 758 Class Project*, Madison, WI, USA, 2009.

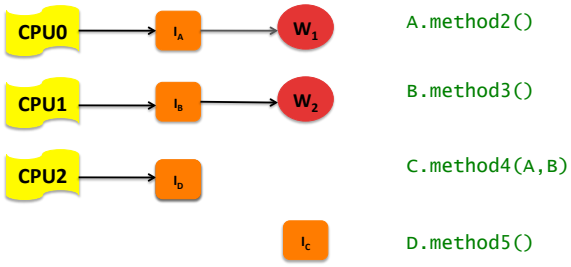
(a) Step1: Methods *2* and *3* are launched into the runtime system in sequential order. Serializers (not shown) corresponding to *A* and *B* are scheduled and the method invocations I_a and I_b are enqueued for execution.



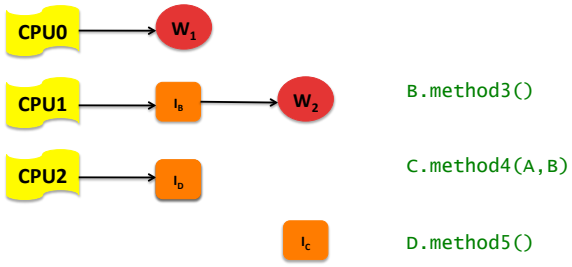
(b) Step2: Method *4* is launched in sequential order. Method invocation I_c (and the serializer) corresponding to *C* is scheduled for execution. Runtime system identifies it as a special method and generates two tokens



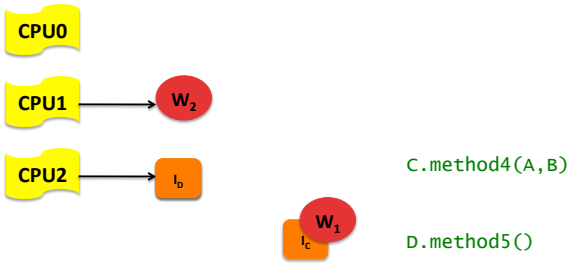
(c) Step3: The generated tokens W_1 and W_2 are queued behind the corresponding invocations of *A* and *B*. Serializer (including the serializer and invocation I_c corresponding to *C* is shelved.



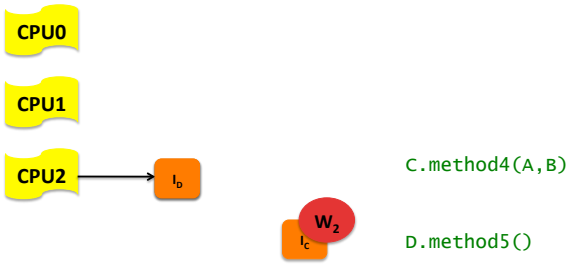
(d) Step4: Method *5* is launched in sequential order by scheduling the invocation I_d corresponding to *D* for execution.



(e) Step5: I_a completes execution. Token W_1 is scheduled for execution.



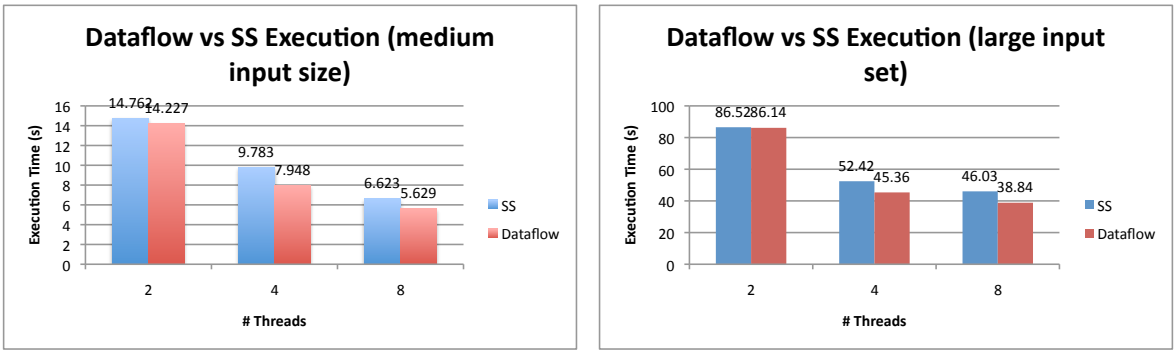
(f) Step6: W_1 is returned back to the owner I_c. In the mean time I_b completes execution. Serializer for *A* descheduled from the runtime system.



(g) Step7: W_2 is returned back to the owner I_c. Serializer for *B* is descheduled from the runtime system.

(h) Step8: Once I_c receives all the tokens, it is scheduled for execution on an available thread.

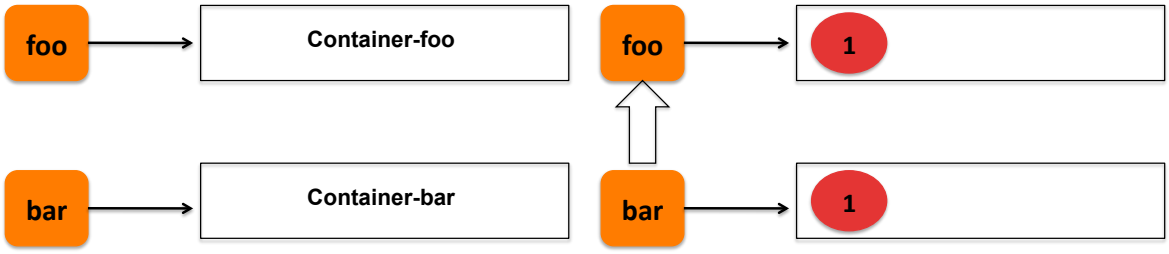
Figure 4: Mechanics of dataflow execution.



(a) SS vs dataflow on medium input set.

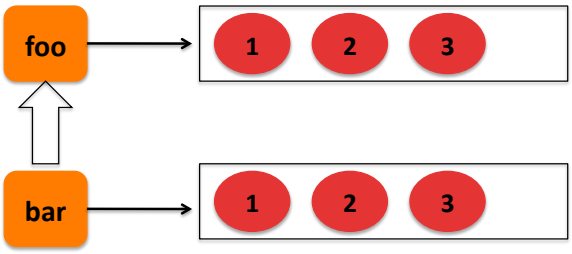
(b) SS vs Dataflow on large input set.

Figure 6: Performance comparison of SS and dataflow on medium and large input sets.



(a) Step1

(b) Step2



(c) Step3

Figure 8: Mechanics of SS-GPU.