A Model for Programming Large-Scale Configurable Computing Applications

Carl Ebeling*, Scott Hauck†, Walter Ruzzo*, Corey Olson†, Maria Kim†, Cooper Clausen*†, Boris Kogon*
*Dept. of Computer Science and Engineering and †Dept. of Electrical Engineering
University of Washington
Seattle, WA 98195

***Abstract*—It is clear that Exascale computing will require alternative computing substrates such as FPGAs as an adjunct to traditional processors to stay within power consumption constraints. Executing applications, or parts of applications, using specialized, fine-grained computing structures configured into FPGAs can achieve a large increase in both performance and energy-efficiency for many important applications. Unfortunately, the process of designing and implementing these specialized hardware structures is tedious and requires expertise in hardware design. The lack of programming models and associated compilers for configurable computing has impeded progress in the use of FPGAs in large-scale computing platforms. This paper describes a parallel programming model and compilation strategy that we are exploring as a way to describe large-scale applications using familiar concepts and abstractions, but which can be implemented using large-scale, fine-grained parallel computing structures. We first outline this model and show how it can be used to develop a scalable parallel solution for the genomic short-read reassembly problem.**[1]

Introduction

Our goal is to enable programmers to write parallel applications for large-scale computing systems that include a substantial amount of computing power in the form of FPGA accelerators. What we have in mind as an eventual target are platforms with 10's to 100's of processors and 100's to 1000's of FPGAs. Our approach is to adopt and adapt a set of techniques and concepts that have been developed for programming parallel and distributed computers to solving the problem of implementing large scale algorithms using hardware-accelerated computing platforms. This includes providing a computation model that enables algorithms to be described at a relatively high level of abstraction and then mapped across a large number of computation nodes that include processors, FPGAs, and possibly other accelerators like GPUs. This will allow applications to be partitioned and mapped to a combination of software and hardware resources using a single, consistent programming model.

There have been many efforts to compile high-level languages like C to hardware and these are now becoming widely available. By contrast, our work is focused more on the system level; in fact we implicitly assume the existence of "module compilers" that efficiently map small C programs to hardware. Our model is similar in spirit to several other models that address the system level, particularly TDM-MPI[1] and IBM's Lime[2], but differs in the use of higher-level abstractions for describing parallelism.

Programming Model

Our model is based on an object model: A system is comprised of objects, each containing data and methods that implement the functionality of the object. As will be seen, these hardware objects are different from objects used in object-oriented programming. However, we have chosen to call them objects instead of, for example, components, to highlight that they are more similar to software objects than they are to hardware components. Since our goal is to provide a programming model for describing applications that are compiled into hardware, this paper focuses on the model as it relates to hardware. However, it should be clear that compiling programs using this model to software is fairly straightforward. Thus a system can be built using a combination of hardware and software, and functionality can be moved between hardware and software by specifying whether an object is implemented in software or hardware.

An object comprises a set of methods that implement the functionality of the object, along with the data owned by the object, which may be in registers or memory arrays. In hardware objects, methods are implemented as hardware threads that run continuously. Methods wait to be called and then execute some function using the parameters passed with the method call. Upon completion, the method then waits for another method call.

[1]This research has been supported in part by Pico Computing and the Washington Technology Center.

Although methods can be thought of as executing one call at a time, method execution may be pipelined if data dependencies allow it so that several calls are active concurrently.

Methods interact by calling the methods of other objects. Method calls are by default asynchronous: there is no return value, and the caller continues execution immediately after the call in style similar to active messages[3]. Synchronous calls, with optional return values, are also supported when synchronization is required. An object can thus be viewed as a component that contains multiple concurrent threads that interact synchronously with each other, and share the data owned by the object. Variables can be declared shared and protected using arbiters, or methods can be declared to be exclusive to avoid data races within an object.

All hardware objects in a system are statically allocated, meaning they are created and connected together at compile time. Object constructors are executed at compile-time to construct the system as a hierarchical set of objects that are compiled into hardware components. When the system is started, all the object method threads are activated and all main methods are called: these may simply perform some initialization, or they may also begin the concurrent execution of the application.

Method calls can be viewed abstractly as remote procedure calls[4], [5] that are delivered automatically to the receiving object. The hardware compiler can implement this communication more or less efficiently depending on the location of the caller and callee objects. If the callee is known at compile time and is "close" to the caller, then a simple direct-wired connection suffices. More generally, a protocol like MPI can be used to transfer method calls to the destination. However, in most systems the network can be partitioned into smaller networks based on the static call graph.

Thus far our proposed model is rather modest and while it allows the programmer to partition the application into concurrent objects, it does not address the problem of large-scale parallel implementations. This is done in our model using "dynamic" and "distributed" objects.

Dynamic Objects

Dynamic objects provide the illusion of dynamically allocated hardware objects, which provide a convenient mechanism for managing a large number of parallel objects. Dynamic objects are implemented as a pool of statically instantiated hardware objects that are allocated and deallocated automatically by a dynamic object manager.

Dynamic objects are differentiated using an object ID, which must be a parameter to each of the object's methods. All method calls to a dynamic object are delivered to the dynamic object manager, which forwards it to the right object. The first time that an object method is called with a new object ID, an object is automatically allocated by the manager, and its initialization code, if any, is executed. Thereafter all method calls that map to the same ID are delivered by the object manager to this object. When the object has completed, it deallocates itself by calling the deallocate method of the object manager. We have found that reference counting works well for automatic object deallocation.

Distributed Objects

In most cases, an application can be described using a relatively small number of concurrent objects. Generating a parallel implementation involves partitioning the data and computation and distributing these across a large number concurrent objects. We borrow the idea of "distributions" from parallel programming languages like ZPL[6], Chapel[7] and X10[8] to describe how objects in our model are duplicated and distributed to achieve large parallel implementations. Distributions allow the partitioning and parallelization of an implementation across a large number of nodes to be described concisely and implemented automatically.

Objects are often distributed in conjunction with the partitioning of a memory array. A distribution map function is used to describe how the array is partitioned across multiple "locales" by mapping the array indices to locales. A locale can be a specific FPGA, or perhaps even a part of an FPGA. A distribution causes the original object to be duplicated on each locale and the memory to be partitioned across the locales. Calls to methods of the object that include the distribution parameters (array indices) are automatically sent to the locale with the corresponding memory partition, while calls to methods without the distribution parameters are sent to all the objects using a broadcast call. Array accesses are automatically checked and remapped to access the local memory partition. Although objects are typically distributed by partitioning memory as described, any object can be duplicated by providing a distribution parameter and a distribution map function.

The programmer describes a parallel implementation simply by defining the distribution map functions and the compiler then automatically partitions the memory, duplicates the objects, turns method calls into point-to-point wires, bus transactions or network packets that route each call to the appropriate object. A parallel

implementation can then be tuned by redefining the distributions and recompiling.

To achieve even greater parallelism, memory objects can be replicated. For example, in a very large system, access to memory that is widely distributed may cause the network to be a bottleneck. This can be avoided by replicating the memory in addition to distributing it. Method calls are then routed to the nearest replica, reducing the network traffic.

In highly concurrent systems, almost all objects are distributed. Designing a distributed parallel system requires understanding the implications of a particular distribution on memory and communication bandwidth. Object distributions should be "aligned", that is, distributed so that most communication between the distributed objects is local, ensuring that the only non-local communication used is that which is essential to the computation. With our model, the programmer can focus on describing the functionality of a system in terms of individual object classes separate from describing how those objects are partitioned and duplicated to create a parallel system. The system can be remapped very quickly using a different distribution since the resulting changes are automatically generated along with all the communication, including the allocation of method call invocation to local wires or the general network. In the next section, we present an example of using this model to implement a parallel solution to the short read reassembly problem from genomics.

EXAMPLE: GENOME REASSEMBLY

Next generation sequencing technologies have appeared in recent years that are completely changing the way genome sequencing is done. New platforms like the Solexa/Illumina and SOLiD can sequence one billion base pairs in the matter of days. However, the computational cost of accurately re-assembling the data produced by these new sequencing machines into a complete genome is high and threatens to overwhelm the cost of generating the data[9]. Providing a low-cost, low-power, high-performance solution to the re-sequencing problem has the potential to make the sequencing of individual genomes routine[10].

To simplify somewhat, a DNA sample of the target genome is prepared by slicing several copies of the genome at random into many small subsequences which are 30–70 base-pairs (ACTG) in length. Next generation sequencing machines then "read" the base-pair string for each of these subsequences, called "reads". These new sequencing machines can perform these reads in parallel to generate hundreds of millions of reads in a matter of days. This is done on several copies of the DNA sequence resulting in many overlapping reads, which guarantees coverage of the entire reference sequence and allows a consensus to be achieved in the presence of errors in the reading process.

Genome mapping is done using a reference genome as a guide. The location of each read in the reference genome is first determined, called "alignment," and then all the aligned reads are "spliced" together to generate the target genome. This works since genomes are extremely similar, differing in perhaps 1 in 1000 base-pairs. While many reads will match exactly to a location in the reference genome, many do not, either because there was a read error or a difference between the reference and target genome at that location We next describe a parallel hardware implementation of a short read alignment algorithm using our programming model.

Short Read Alignment Algorithm

Our alignment algorithm finds the best positioning for each read, and is based on the algorithm used by BFAST[11]. The alignment is found in two steps. In the first step, a set of "candidate alignment locations" (CALs) is collected for the read using an index of the reference genome. In the second step, the read is compared to the genome at each of the candidate locations and the location that has the best match to the read is reported. It is important both when finding the set of CALs, and when evaluating the match for each CAL, that the algorithm handle multiple mismatches caused by single nucleotide polymorphisms (SNPs) and read errors, as well as insertions and deletions (indels). Although these occur infrequently, it is these cases that are the most biologically interesting and thus most important to identify.

The first step uses an index of the reference genome, called the Reference Index Table (RIT), which can be constructed offline. This index is a hash table that maps all subsequences of the genome of length N to the set of locations where that sequence occurs[12]. N is typically chosen to be between 18 and 22 base pairs so that statistically most subsequences occur just once in the genome. To find a candidate location for a read, we take a subsequence of length N in the read, called a seed, and look it up in the index. If this subsequence of the read has no mismatches or indels, then this will return the locations in the genome where that part of the read occurs. The set of CALs is formed by doing this lookup for all subsequences of length N in the read. If at least one of these seeds is free of mismatches and indels then the set of CALs will contain the location of the actual alignment.

In the second step, a full Smith-Waterman[13] style algorithm is used to compare the read to the reference at each candidate location, with the best location reported for that read. The Smith-Waterman algorithm is a dynamic programming algorithm for performing approximate string-matching which can be mapped very efficiently to a systolic array implementation in hardware.

A typical short read alignment problem for the human genome involves aligning 200 million short reads, about 10–20 TB of data, to a genome of 3 billion base pairs. The size of the index table is about 20GB, and the size of the reference genome is 1–4 GB depending on the data representation. Performing an alignment in software using a computing cluster takes about 6 hours.

Figure 1 shows this algorithm mapped to objects using our model. The operation of each of the objects is described below.

a) Reads: The Reads object runs on the Host. It reads the short reads from bulk storage and calls Seeds:nextRead() with each short read.

b) Seeds: The nextRead() method takes the short read, passed as a parameter, extracts all the seeds one at a time, calling the RIT:nextSeed() method with each seed. Before it does this, it calls Filter:nextRead() with the read and the number of seeds to expect. This is a synchronous call, so the Seeds:nextRead() method blocks until the Filter object is ready for the next read. This synchronization is required because Filter must complete processing of the previous read before we can proceed with the next read. This reduces the amount of parallelism that is achieved at this point in the pipeline, but we assume that this does not affect the overall performance as the bottleneck is in the Smith-Waterman step, which occurs later in the process.

c) RIT: The RIT object owns the index table. When the nextSeed() method is invoked with a seed, it looks up the seed in the index table, which contains the list of CALs for that seed. For each seed it calls Filter:numCALs() with the number of CALs that will be sent for the seed, and Filter:addCAL() with each of the CALs.

d) Filter: The purpose of the Filter object is to collect all the CALs for a short read, and pass along the unique CALs for comparison by the Smith-Waterman units. The nextRead() method is called by Seeds with each new short read. This is a synchronous method call which waits until the previous read has been completely processed. It then initializes the Filter object for the new read, calls SWDispatch:nextRead() with this new read, and then returns, allowing the Seeds:nextRead() method to start generating the seeds for the read. This

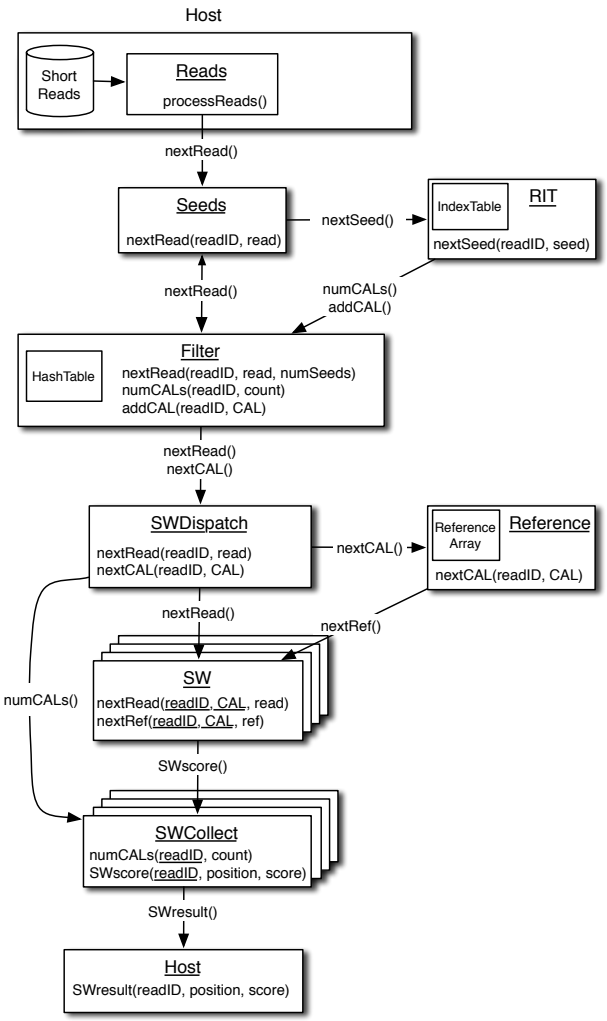

Fig. 1. This call graph shows the short read alignment algorithm mapped into objects in our model. Each block is an object, with the memory and methods indicated in the block. The arrows indicate the calls made by each object, with synchronous method calls shown with a double arrow.

synchronization is required since Filter can only process a single short read at a time.

Each call to addCAL() inserts a CAL into the hash table. If the CAL is not already in the hash table, a call is made to SWDispatch:nextCAL() with the CAL. In this way, all unique CAL values are forwarded to the SWDispatch object. The Filter object knows when the last addCAL() method call has been received for a read by counting the number of seeds (numCALs() method calls) and the number of CALs per seed (total count in the numCALs() method calls).

e) SWDispatch: The SWDispatch object organizes the use of the Smith-Waterman (S-W) dynamic ob-

jects. Each call to nextCAL() causes one call to SW:nextRead(), passing the short read to the S-W object, and one call to Reference:nextCAL() which requests the Reference object to forward the reference string at the given CAL to the S-W object.

f) Reference: This object contains the reference genome data. When given a candidate location by the nextCAL() call, it reads the section of the reference at that location and calls SW:nextRef() to send it to the S-W object.

g) SW: The S-W object is a dynamic object, indicated by the multiple boxes in the figure, which means there are many static instances of the object that are managed by an object manager. Both the nextRead() and nextRef() methods use the (read ID,CAL) pair as the dynamic object ID. Using a dynamic object for the Smith-Waterman unit allows many comparisons, which take a relatively long time, to proceed in parallel. An S-W unit is allocated automatically by the first method that arrives with a new object ID, and when the comparison is finished and the score reported via the SWCollect:SWScore() method, it deallocates itself by calling the deallocate method of the object manager. S-W uses a deeply pipelined dynamic programming algorithm, which means that it can deallocate itself when it is ready to start performing a new comparison, which is *before* it has completed the current comparison. The new comparison is then overlapped with the previous comparison.

h) SWCollect: This object collects the scores reported for all the different CALs for a short read. This is also a dynamic object which is keyed by the readID parameter. By making this object dynamic, the Smith-Waterman units can be working on more than one short read at a time. The SWCollect object keeps track of the best scores reported, and when all scores have been reported, it calls Host:SWResult() with the best results and deallocates itself. SWCollect uses reference counting for deallocation: numCALs() is called with the number of scores to expect.

Synchronization

It is worth noting that there are two types of synchronization used in this example. First, there is an explicit barrier synchronization used in the first part of the pipeline, where the Seeds object must wait for the previous read to be completely processed by the Filter object before it starts processing the next read. This ensures that the short reads pass through the objects in order: All method calls associated with one read are made before method calls for the next read are made. This of course reduces parallelism. The second type of synchronization is enabled by dynamic objects and reference counting. These objects are allocated implicitly and method calls can be interleaved arbitrarily since they are automatically directed to the appropriate object. Reference counting allows objects to automatically deallocate themselves when all expected method calls have been received.

Parallel Implementation

There is already substantial parallelism in the implementation as described, particularly with the many concurrent, pipelined Smith-Waterman units implemented using dynamic objects. To increase the performance of this implementation, we need to consider where the bottlenecks occur. If we assume that we can always make the pool of S-W objects larger, then the bottleneck occurs at the RIT and Filter objects. For each read, Seeds makes a large number of calls to RIT:nextSeed(), each of which is a random access into the index table which must be stored in DRAM. We can increase the performance of the RIT by partitioning and distributing it across multiple nodes so that multiple accesses can proceed in parallel.

This now moves the bottleneck to the Filter object, which now gets multiple simultaneous addCAL() method calls from the concurrent RIT objects. We can remove this bottleneck by duplicating the Seeds, Filter and SWDispatch objects using an "aligned" distribution. In other words, we duplicate and distribute these objects so that each Seeds object communicates with one Filter object, which communicates with one SWDispatch object, all of which are handling the same read. Assuming that reads are processed in order by readID, using the low-order bits of the readID to do the distribution uses these objects in round-robin order.

At this point, the Reference object becomes the bottleneck, as several SWDispatch objects call Reference:nextCAL() concurrently. This can be solved by partitioning and distributing the Reference memory. Assuming that the CALs are spread out more-or-less uniformly, this enables multiple accesses to proceed concurrently. In the limit, we can reach the point where the Reads object becomes the bottleneck and we can process short reads as fast as we can send them to the FPGA accelerator.

Of course, we have described this implementation using an idealized view of the hardware platform. In practice, there will be only a small number of large memories connected to an FPGA, and thus partitioning the RIT and the Reference into many memories for concurrent access can only be done by spreading the implementation across a large number of FPGA nodes. One option is to duplicate and distribute the "main" objects (all except for RIT and Reference) across the

FPGA nodes using an aligned distribution based on readID. This keeps the method calls between them local to an FPGA. However, the method calls to the RIT and Reference are non-local because neither of them can be partitioned by readID. Beyond some degree of parallelism, the communication implied by these calls will become the bottleneck since it is all-to-all communication.

An alternative way to distribute the objects is to use a distribution based on CALs. This partitions the Reference and the RIT, but it means that the remaining "main" objects are replicated instead of distributed. That is, each short read is processed by all of the replicated objects, so that the Seeds:nextRead() method call is broadcast to all the replicated copies of Seeds and remaining method calls are made on the local object copy.

Distributing/replicating by CAL means that each RIT object only has the CALs assigned to its partition, and thus the objects only handle a subset of all the CALs. This means that all of the communication between objects is local, and so there is no inter-FPGA communication except for the initial broadcast. However, this partitioning has consequences both on the algorithm and the performance. First, since each Filter object sees only a subset of the CALs, it cannot filter CALs based on information about all the CALs for a read, and so we may have to process more CALs than necessary. Second, if we partition the objects too finely, then many partitions may have no CALs at all for a read, and the objects will spend a lot of time doing nothing. So, for example, if there are an average of 32 CALs per read, we will start to lose efficiency if we partition much beyond a factor of 8.

We can, however, easily combine replication with partitioning and distribution. For example, we could partition and replicate by 8 first, and then within each partition distribute objects by another factor of 8. This partitions each RIT and Reference (by readID) by only a factor of 8, and the resulting communication bandwidth should not overwhelm the network.

Conclusion

This paper has outlined our vision of a new programming model that will allow programmers to quickly design new applications targeting large-scale reconfigurable computing systems. We have tried to describe, by means of a real example, how this model can be used to explore and implement many different parallel implementations. We see no real barrier to compiling objects in this model to software, which would allow implementations to be prototyped in software and then moved to hardware once the design has been refined. This also facilitates co-design since both software and hardware objects would have the same abstract interface.

There is a substantial amount of work left to do to make this a usable model. This includes formalizing the language, understanding what other features are needed, developing a compiler that can map objects to hardware using our dynamic and distributed object model in combination with state of the art "C-to-Hardware" compilers, and compiling method calls efficiently using a variety of different communication mechanisms. We hope that our research will take a large step towards enabling large-scale reconfigurable computing.

References

[1] M. Saldana, A. Patel, C. Madill, D. Nunes, D. Wang, H. Styles, A. Putnam, R. Wittig, and P. Chow, "MPI as an abstraction for software-hardware interaction for HPRCs," in *Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications. HPRCTA 2008.*, Nov. 2008, pp. 1–10.

[2] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: a java-compatible and synthesizable language for heterogeneous architectures," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 89–108.

[3] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," in *ISCA*, 1992, pp. 256–266.

[4] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39–59, 1984.

[5] J. Maasen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman, "Efficient Java RMI for Parallel Programming," *ACM Transactions on Programming Language Systems*, vol. 23, no. 6, pp. 747–775, 2001.

[6] B. Chamberlain, S.-E. Choi, C. Lewis, C. Lin, L. Snyder, and W. Weathersby, "ZPL: A Machine-Independent Programming Language For Parallel Computers," *IEEE Transactions on Software Engineering*, vol. 26, no. 3, pp. 197 –211, March 2000.

[7] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, 2007.

[8] 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*. New York, NY, USA: ACM, 2005, pp. 519–538.

[9] J. D. McPherson, "Next-generation gap," *Nature*, vol. 6, no. 11s, 2009.

[10] E. R. Mardis, "The impact of next-generation sequencing technology on genetics," *Trends in Genetics*, vol. 24, no. 3, pp. 133 – 141, 2008.

[11] N. Homer, B. Merriman, and S. F. Nelson, "Bfast: An alignment tool for large scale genome resequencing," *PLoS ONE*, vol. 4, no. 11, p. e7767, 11 2009.

[12] D. S. Horner, G. Pavesi, T. Castrignan, P. D. D. Meo, S. Liuni, M. Sammeth, E. Picardi, and G. Pesole, "Bioinformatics approaches for genomics and post genomics applications of next-generation sequencing," *Briefings on Bioinformatics*, vol. 11, no. 2, 2010.

[13] M. S. Waterman and T. F. Smith, "Rapid dynamic programming algorithms for rna secondary structure," *Advances in Applied Mathematics*, vol. 7, no. 4, pp. 455 – 464, 1986.