Cascade: A Parallel Programming Framework for Video Game Engines

Andrea Tagliasacchi
School of Computing Science
Simon Fraser University
Burnaby, BC, Canada
ata2@cs.sfu.ca

Ryan Dickie
School of Engineering
Simon Fraser University
Burnaby, BC, Canada
rdickie@sfu.ca

Alex Couture-Beil
School of Computing Science
Simon Fraser University
Burnaby, BC, Canada
asc17@sfu.ca

Micah J Best
School of Computing Science
Simon Fraser University
Burnaby, BC, Canada
mbest@sfu.ca

Alexandra Fedorova
School of Computing Science
Simon Fraser University
Burnaby, BC, Canada
fedorova@cs.sfu.ca

Andrew Brownsword
Electronic Arts BlackBox
Vancouver, BC, Canada
brownsword@ea.com

ABSTRACT

The Cascade Parallel Processing Framework (PPF) is a user level library that facilitates manual parallelization of complex C++ systems. In Cascade, processing duties of the system are enclosed in a *Cascade Task*. Tasks are linked by dependencies in a *task dependency graph*. The task graph is traversed at runtime by the *Cascade Job Manager* who assigns tasks to threads for execution. The Job Manager must correctly satisfy dependencies while maximizing performance. While a task-based PPF is not a new concept, Cascade's unique goal is to address complex systems, such as video game engines. These systems are built as multiple interacting sub-systems, with non-trivial dependencies. Existing PPFs, while suitable for parallelization of individual sub-systems, do not solve the entire problem. In this paper we describe the early design and implementation of Cascade, present preliminary evaluation, and outline plans for future research.

1. INTRODUCTION

Cascade is a new parallel programming framework for complex systems. With Cascade, the programmer explicitly structures her C++ program as a collection of independent units of computation, or tasks. Tasks are linked by dependencies of various forms. For example, when one task produces data for another task, there is a *dataflow dependency* among them. Dependencies link tasks in a directed acyclic graph, a *task dependency graph*. The Cascade Job Manager traverses the graph at runtime and assigns tasks to threads for execution. This graph captures both the logic to be executed and the flow of the data through the system. As any circular dependency would make execution impossible an acyclic graph is necessary.

This explicit task-based programming style is targeted for applications where work cannot be split into parallel units of computation automatically (by a compiler) or semi-automatically (e.g., using parallel loop directives). Cascade is designed for applications that contain a number of distinct subsystems that have a high amount of inter-subsystem dependencies which are well known by the developer, thus, the Cascade system simply gives the power to exploit them.

Video game engines are structured as multiple interacting sub-systems: physics, artificial intelligence (AI), geometric computation, scene management, graphics, I/O, and so on. In these sub-systems, in addition to dataflow dependencies among tasks, there can also be *interactive dependencies* (occurring when tasks share data) and *real-time dependencies* (occurring when a task must be completed by a certain deadline). Cascade allows the programmer to express these additional dependencies, and this helps the Job Manager extract the most parallelism out of the workload and achieve qualitative performance objectives.

Cascade allows layering of other parallel constructs, such as OpenMP parallel for-each loops [2] or *map/reduce* [11], on top of its task-based framework. While the entire game engine cannot be parallelized using those simpler constructs, individual sub-systems can. By layering other parallel constructs on top of Cascade, the sub-systems can run in parallel and share the available hardware in an effective and low-overhead fashion. Cascade Job Manager juggles tasks from these separate task trees, satisfying dependencies and extracting maximum parallelism.

Parallelization of game engines is a timely and impor-

tant problem, because video game consoles, like other hardware, are becoming predominantly multicore: Sony PlayStation 3 now ships with IBM's multicore Cell Processor and Xbox-360 comes with three multi-threaded cores. Existing parallel programming frameworks could be used to parallelize individual sub-systems of a game engine, but do not solve the entire problem. In building Cascade, we hope to create a research platform for investigation of this problem. Cascade has been designed to allow the expression of the interrelationships between these sub-systems to facilitate the parallel scheduling of sub-tasks in way that respects these relationships.

The rest of the paper is organized as follows. In Section 2 we describe the structure of a typical game engine and present our vision of an ideal PPF for this application domain. In Section 3 we describe the preliminary implementation of Cascade, and in Section 4 we report some performance results and case studies. In Section 5 we discuss related work, and in Section 6 we summarize.

2. A PPF FOR GAME ENGINES

The state of the video game is captured by the *game scene* and a collection of *game objects* that are re-drawn on the screen in each video frame. From one frame to the next, objects' position and appearance are transformed based on the user input. These transformations are computed and presented to the user by various subsystems: physics, AI, scene management, graphics, audio, and so on. A single object is handed from one sub-system to the next: for example, to transform a game character, its skeletal position is first updated by the physics and AI subsystems, and then "skinning", texture and lighting are applied by the graphics subsystem.

Usually each sub-system performs a series of identical operations on a large collection of game objects. These batches of computation are essentially data-parallel iterations over ordered collections, and so they can be parallelized either via explicit task-based programming or using existing high-level PPFs targeted at iterated collections (parallel for-each loops, *map/reduce* [11], StreamIt [9], Ct [3], etc.).

Our vision is that in Cascade, higher-level PPFs may be layered on top of the Job Manager. For example, StreamIt and Ct might be used to implement task trees that are then scheduled alongside trees from other subsystems (which also might be implemented with higher level tools). The nice thing about running multiple independent trees in parallel is that they could be unordered with respect to each other and thus have lots of potential parallelism. And yet they can be made dependent upon each other because they share the underlying job manager. We can envision a StreamIt task tree that doesn't begin until the Ct task tree it is dependent on finishes, and that one might be dependent on another system's parallel-foreach loop and another StreamIt job.

Very complex systems with lots of parallelism can be assembled using this approach. To do all this, the higher level systems would want to expose some kind of handle to their internally created task so that the programmer can use that to create dependencies. For example, the programmer may specify that some tasks in the AI sub-system's tree depend on some (but not all) tasks in the physics sub-system's tree, a so-called *partial dependency*. The Job Manager will then be able to begin scheduling tasks from the AI tree before the physics tree has finished running. This way more parallelism can be extracted out of the workload than with explicit threaded implementation or with higher-level tools alone.

In exploring the viability of this vision, we expect to address many research problems. One problem has to do with task dependencies. As we mentioned, task dependencies can arise in a variety of forms: dataflow dependencies, interactive dependencies, real-time dependencies and possibly others. Interesting work remains to be done in the area of defining each kind of dependency and in refining the Job Manager to allow their explicit expression. As an example real-time dependencies are one key type of dependency for our chosen target domain, video games. It is crucial to the requirements domain to that our architecture has the facilities to specify that a task is dependent on a system event and that time budgeting can be explicitly specified. We expect that modeling real-time events and constraints will form a major part of the final version of Cascade.

Another interesting problem is amortizing scheduling overhead. If the task is too small, the running time may be dominated by the overhead of creating and scheduling this task. Therefore, it may be wise to batch several smaller logical tasks, called *chores*, in a single Cascade Task. In this respect, high-level parallelization constructs layered on top of Cascade could be useful, because some of them already implement batching.

Another set of interesting problems has to do with catering to the architectural properties of the system. Existing game consoles consist of heterogeneous cores (Sony PS3 comes with several vector cores and a single general-purpose core). In the future, even general purpose cores will be heterogeneous (they may expose the same ISA, but will differ in features and performance). The Job Manager will face the challenge of assigning the tasks to the "right" cores, to achieve the optimal performance/power trade-off. Another example of such *architecture-specific optimizations* has to do with cache utilization. Tasks working on the same data could run faster if they are assigned to cores that share a cache. If, on the other hand, the instruction cache is scarce, tasks running the same code should be placed on adjacent

cores (or thread contexts).

These, and other related problems, are similar to those in operating systems schedulers on multicore systems [4][7][18]. Operating systems typically try to determine the best assignment of threads to cores via online performance monitoring, because they have no knowledge of threads' characteristics. In our environment, the Job Manager (that acts as scheduler) is part of the application, so the software can provide semantic hints to simplify scheduling. The addition of semantic information to a scheduler not only relieves it of having to do dynamic measurement, it also makes information available that may simply not be discernible at all. This is particularly true when the OS and hardware see only threads, and not the task division/organization that sits on top of the threads. Also, any observational/measurement system is based on the idea that by looking at what *has* happened it can predict what *will* happen, but this does not always work in highly dynamic and changing systems like games. The system itself will often have fore-knowledge of what it is going to do, and can tell the scheduler by hinting.

In summary, Cascade framework must provide an easy way for the programmer to express task dependencies so as to allow the Job Manager to extract the maximum parallelism, and Cascade Job Manager must juggle tasks so as to amortize scheduling overhead and cater to the underlying architecture. This discussion only scratches the surface of the interesting research problems that arise in this environment; we omit a broader discussion due to space constraints.

3. CORE ARCHITECTURE

At the heart of our Cascade is the Job Manager. It is built on top of pthreads (on Unix) or WinApi (on Windows). We used boost::function and boost::bind provided in the Boost C++ library. boost::function is a C++ struct with an overloaded operator(). It wraps the function arguments and function pointer, allowing for an easy way to convert functions to Cascade Tasks.

The Job Manager supports both static and dynamic task graphs. The static version is designed for applications where the dependency graph is built all at once ahead of execution, as in scientific computing. The dynamic version allows for tasks to be constructed and appended to the graph at runtime in order to meet the dynamic needs of game development. Both versions use lock-free synchronization. In the future, we will support hybrid graphs, where a static graph can be added to the dynamic graph. This will give us much of the best of both worlds, particularly if a static portion can be re-submitted at each frame with a minimal amount of work.

The Job Manager recursively executes the collection of tasks starting from the root of the dependency graph. In order to traverse the graph correctly, the Job Manager maintains two lists. The first, *runList*, contais tasks that are currently running or have recently completed. The second, *waitList*, contains tasks whose dependencies have been satisfied and that are ready to be scheduled. Tasks in the runlist are periodically checked for the presence of runnable children. If the child is runnable it is placed in the waitlist.

There are two modes for traversing the *waitList*: LIFO and FIFO. The LIFO mode executes first the children of recently completed tasks, and in doing so keeps the cache reasonably warm. FIFO schedules earlier tasks first, and in doing so may be better at meeting real-time deadlines.

Tasks from *waitList* are emptied into *ThreadPool*, a lightweight class designed to assign tasks to some collection of threads. It contains a single thread for each virtual processor (i.e., CPU, core, or hardware context).

Proper memory allocation in the Job Manager is crucial for performance, because memory management can involve expensive system calls, synchronization, and fragmentation. In the static case, we will need to rapidly allocate task nodes in a cache friendly manner. In the dynamic case, *Task* nodes may be created by any of the running threads with real-time constraints. In both cases, objects also need to be freed rapidly. In order to address these problems, we create a memory pool for Task objects. The dynamic version keeps a fixed sized pool of objects and automatically reclaims tasks that are no longer needed. This process is lock-free and wait-free and has minimal overhead.

The Task node provides an excellent place to store profiling information and hints to the scheduler. Each task node contains a user supplied unique identifier along with optional timing parameters to track the state changes of the task node. We have developed graphical tools in order to visualize this output. Presently, this data is only for the user but in the future we will consider using this information to automatically optimize the dependency graph and its execution.

We have also implemented several parallel constructs layered on top of Cascade: map, reduce, zip, and others. We omit description of these components due to space constraints.

4. BENCHMARKS AND CASE STUDIES

In our preliminary evaluation of Cascade, we consider its effectiveness from two points of view, those of performance and programmability. To benchmark the performance we chose sequence alignment, a dynamic programming algorithm heavily used in bioinformatics. In presenting this benchmark we are not attempting to show a better parallelization scheme for this algorithm, as extensive research has already been conducted. Rather, we want to show quantitatively

that overhead introduced by our task-based approach is competitive with more traditional multithreading multithreading-based solutions in terms of performance. The relative simplicity of this algorithm allows for more objective comparison. We believe that this serves to illustrate the flexibility of our approach in that the system performs well not only a very broad decomposition of subsystem, such as the video game case study below, but also those that use a high degree of granularity of their task decomposition.

To evaluate programmability we modified "Destroy The Castle (DtC)" [1], a game demo from Intel. Parallelization of game engines is a relatively new topic and has distinct challenges from those found in operating system or 'hard' realtime systems. Robust general solutions tailored to the domain do not exist yet. Cascade is, as far as we are aware, the only research-oriented architecture to use this form of task decomposition and expression. Solutions exist in the commercial world, but they are proprietary and generally tightly coupled to the software they are being used to parallelize. As we already highlighted, our system is a particularly well suited to game engines which are a collection of distinct subsystems where each subsystem can co-exist in parallel with several others while needing to be exclusive of the rest. These subsystem map naturally on to our task-based model and have high performance demands requiring that any management scheme have low and consistent overhead.

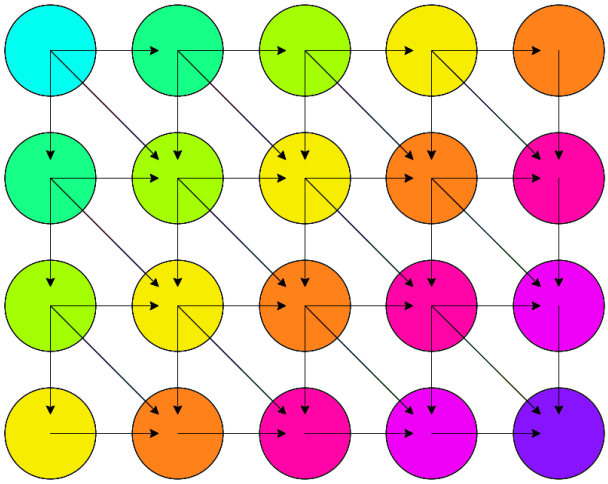
4.1 Performance: Dynamic Programming, Sequence Alignment

The sequence alignment algorithm computes optimal alignment of two strings. In bioinformatics, sequence alignment is used to identify regions of similarity of DNA or RNA sequences. To compute the alignment of two strings of size n and m, the algorithms creates an $n \times m$ matrix, where rows and columns correspond to the strings' characters. The aligner computes scores for each element (i, j) in the matrix based on maximum score in adjacent elements $(i-1, j), (i-1, j-1), (i, j-1)$. So the computation of each score depends on the scores of three neighbors (see Figure 1). Using this dependency information, the programmer constructs the Cascade task graph, starting with element $(0, 0)$ before continuing to adjacent elements (see Figure 2).

To transform this information into an OpenMP implementation, the task dependency graph must be manipulated into a list of task-sets that can be run in parallel:

$\{(0,0)\}, \{(1,0),(0,1)\}, \{(2,0),(1,1),(0,2)\}, ..., \{(n,m)\}$

Each set is a collection of diagonal elements that can be run in parallel; however, since each diagonal set is dependent on the previous set, the diagonal must complete before calculating the next. In contrast, the Cascade implementation does not have such an artificial constraint, and will keep idle threads busy by progressing to subsets of other diagonals.



(a)

Figure 1: Data dependency of matrix elements. The arrows indicate the true data dependencies, which can be used with the dataflow framework. The diagonal groups represent elements that can be calculated in parallel once the previous diagonal group has been calculated. (that's the OpenMP way).

The benchmarks are run on a Sun T2000, 64-bit SPARC v9 processor with 32 hardware thread contexts on eight cores. The sequence aligner computes the optimal alignment of two strings both of length 7700, thus requiring 59,290,000 element calculations. To minimize system overhead incurred from switching tasks, the matrix is subdivided into square blocks whose elements are sequentially calculated. Even with optimal block sizes, the Cascade implementation completes 1.5 times quicker than the OpenMP implementation. See Table 1 for a break down of execution times. We also collected data showing that Cascade runs faster despite the scheduling overhead, because it extract more parallelism out of the workload and produces less thread-idle time than OpenMP.

This result demonstrates the efficiency of our preliminary implementation and the fundamental scaling potential of the task-based approach. The performance achieved by this task-based approach is competitive with a classical OpenMP implementation. In addition, the task-based PPF extracts greater parallelism out of the workload than a higher-level parallel construct thanks to having precise information on dataflow dependencies.

In a more subjective vein, we discovered in producing

```
JobManager jm( NUM_THREADS );
_t = new Task*[ _job_matrix_size ];

//Create the tasks in the array
for( int y = 0; y < _rows; y++ ) {
  for( int x = 0; x < _cols; x++ ) {
    _t[ index( x, y ) ] = jm.createTask(
      boost::bind( calcElement, x, y ) );
  }
}

//Set task at [0,0] as the entry point
jm.getRoot()->addChild( _t[0] );

//Set up all remaining dependencies
Task *t;
for( int y = 0; y < _rows; y++ ) {
  for( int x = 0; x < _cols; x++ ) {
    t = _t[index( x, y )];
    if( y < _rows- 1 ) {
      //add bottom child
      t->addChild( _t[index(x, y+1)] );
    }
    if( x < _cols - 1 ) {
      //add right child
      t->addChild( _t[index(x+1, y)] );
      if( y < _rows - 1 ) {
        //add bottom right child
        t->addChild( _t[index(x+1, y+1)] );
      }
    }
  }
}
```

Figure 2: Cascade task graph creation for sequence alignment.

this test that a task-based decomposition of the problem was easier to implement than one using OpenMP.

4.2 Programmability: Game Engine, Destroy The Castle

Given that Cascade is primarily designed to model complex systems such as video games it was necessary to assess how well it was suited for parallelization of these systems. For this purpose we modified the "Destroy The Castle (DtC)" [1] video game, a technology demo from Intel. DtC proved to be well suited to our purposes as it contained many of the facets of a modern game engine, but in a simplified form, which allowed us to test our design assumptions. In this section we report on our preliminary experiences of converting DtC to use Cascade and describe the lessons learned in the process.

The existing DtC code used Thread Building Blocks and standard Windows threading primitives to coordinate the various tasks that produce a frame. Once player input had been taken into account the physical behaviors of all the non-intelligent entities would be calculated. The AI system would then be invoked using the current physical state of the world to make decisions for the AI controlled entities. Finally once the current position of all entities had been determined the particle system would be invoked to add atmospheric effects such as smoke. We converted this code to use the Cascade framework. The resulting dependency structure is shown in Figure 3.

Refactoring the existing code to use Cascade was, in general, straightforward. Sequential function calls and various types of 'wait' calls were mapped onto dependencies. For most of the modules only their points of invocation needed to be modified and their internals were left virtually unchanged. One large issue was encountered when dealing with DirectX, Microsoft's DirectX API for graphical and audio output. DirectX runs its own threads internally, and those threads are outside of control of Cascade's Job Manager. The inability to control this sub-system limited the parallelism that could be extracted and created counterproductive interactions with Cascade threads. This kind of 'external dependence' issues should be addressed in future versions of Cascade and should only affect existing systems that use closed components.

With the exception of this conflict between high level process management systems the transformation of DtC to use Cascade was not at all arduous. While further case studies of this type are needed to determine if this simplicity is as common we suspect it has given us confidence in approach that we are taking. While the code contained in DtC is admittedly simpler than a commercial engine we were able to carry out the conversion with no previous knowledge of the code and a very limited time budget.

Block Size	Cascade time	OpenMP time
50×50	1.535	2.418
75×75	1.358	2.405
100×100	**1.348**	2.292
125×125	1.406	2.103
150×150	1.459	2.199
175×175	1.524	2.323
200×200	1.581	2.466
250×250	1.763	**2.027**
300×300	1.919	2.133

Table 1: Execution time in seconds depends on the framework, and number of elements per block. A larger block size results in fewer tasks. Finding an optimal balance between the level of parallelization, and framework overhead requires static profiling.

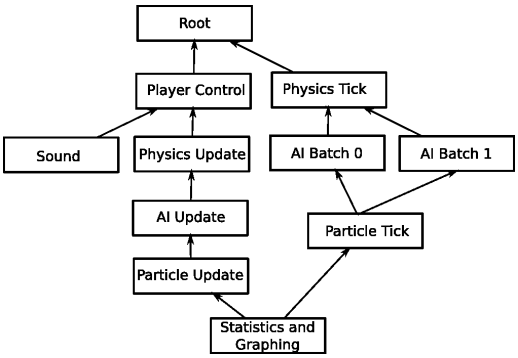


Figure 3: Cascade task graph for calculating a frame of DtC.

While a complete 'post-mortem' of the refactoring of this very specific piece of software would not be very instructive to the reader something of the process may help to illustrate some of the attributes of Cascade.

The code to perform the game logic, or simulation, was originally all invoked from a single lengthy function encompassing calls to all non-rendering related libraries and objects. This monolithic function was first divided into many separate functions which became the tasks. So, in the first iteration task graph began as single path, mimicking the serial nature of sequential code. No parallelism was present at this point, but with only some minor refactoring and not more than 30 lines of additional code Cascade was present and the code was executable.

Following this was the process of exploiting parallelism over several iterations. Our architecture was not designed for automatic parallel extraction, but for the easy expression of the knowledge of the designer. In this case we first determined that the code responsible for sound was in actuality independent from every operation except those involved with determining the input of the player and so it was made a child of Player Control task with no dependents of its own.

In a similar fashion with minor source code changes the tree widened and some tasks were split into smaller tasks. This task splitting shows one of Cascade's strengths. The sequence alignment tests showed that Cascade was capable of handling many tasks with minimal overhead. The designer has the option to employee a high degree of granularity for some systems and very course granularity for other depending on amount of potential parallelism without incurring too much overhead in either case.

A lot was learned from this implementation experiment. The lessons that will certainly inform the growth of Cascade are the need for a 'distributed' scheduling mechanism, the importance of batching, and the need for the specification of partial dependencies.

In the early implementation of Cascade the scheduler was implemented in its own thread that was invoked upon completion of every task. While this caused no performance problems with the sequence alignment benchmark, with DtC it produced a great deal of unnecessary context switching and significant performance overhead. As a countermeasure, we changed the scheduler to execute in the context of Cascade threads. The scheduler now runs in a distributed fashion as a postscript to each task. This new design significantly reduced the overhead and produced smoother performance.

Another lesson learned was the importance of batching and the need for specifying partial dependencies. Consider, in Figure 3, the dependence that the Particle system has on the AI system. In this case it is not required that *all* calculations done by the AI system complete before the Particle system begins. Once the AI system completes modification for a single entity, any particle effects for this entity can already be calculated. So in theory the Particle system could start significantly earlier than the completion of the AI system. In the extreme case, the dependency graph could be built of fine-grained tasks, each performing an update on a single entity. We found, however, that such fine-grained parallelization was inefficient due to scheduling overhead. Instead, batching of multiple updates in a single task dramatically improved performance. On the other hand, having very large batches limits the parallelism, so it would be desirable to create moderately sized batches and specify partial dependencies between the batches belonging to the AI and the Particle systems. To find a happy medium, future versions of Cascade will be designed with a mechanism to reflect these partial dependencies.

At the present, we are continuing the transformation of DtC to Cascade, and hope to report more on our experiences as well as performance results in the future. Also, we are examining other platforms, which has more complexity than DtC, to apply cascade to for a further case study. The will allow us to further evaluate and refine our architecture and give us further confidence in generality of our assumptions.

5. RELATED WORK

Increasing parallelism in the hardware coupled with the difficulty and limitations of thread-based programming has inspired much new research in parallelization techniques, ranging thread-level speculation (TLS) to new parallel programming languages. We compare and contrast Cascade with some of these techniques. Due to space limitations we are unable to provide a complete discussion.

Solutions like OpenMP [2] and TLS [19] are suitable for parallelization of portions of code, such as loops, and are not whole-system solutions. They can be integrated with Cascade for parallelization of individual tasks.

Use of new programming languages with explicit support for parallelism presents an attractive alternative [5, 6, 10, 12], but the downside is that this requires retraining of engineers. A notable exception is *Cilk++* [8], a parallel syntax extension to standard programming languages like C++ that allows specifying synchronization dependencies between functions. Cilk, unlike Cascade, does not allow explicit constructions of task graphs and permits no modifications to the Job Manager.

Recently, the *map/reduce* paradigm [11] has received much attention thanks to its ability of semantically describing potential concurrency in an algorithm. Operators (map/reduce), which act on a collection, hide the

complexity of parallel implementation. There are several PPFs that use *map/reduce*. One example is the Galois system [14]. This framework, which also has a job manager, has recently been extended with several different task scheduling algorithms tailored for different workloads. The authors found that catering the algorithm to the workload results in significant performance improvements [13]. Some of the lessons learned in that study will be applicable to our research.

Other recent PPFs that use *map/reduce* paradigm are the Ct framework from Intel [3], RapidMind [16], and Merge [15]. They allow inserting an abstraction layer over standard C++ to exploit parallelism in a map/reduce fashion over heterogeneous hardware. RapidMind performs just-in-time (JIT) compilation to create platform-specific code. Merge allows the specification of multiple implementations at design time and chooses the right one at runtime. These high-level tools, perhaps with the exception of RapidMind, can be integrated with Cascade. RapidMind performs JIT compilation, which is a limitation in soft real-time systems like game engines.

Even though all the discussed approaches can be sufficient in representing the most common types of parallelism, there are cases in which we want to clearly express *complex dependencies* between parallel tasks. StreamIt [9] and *Intel Thread Building Blocks* (TBB) [17] are task-based PPFs suitable for this purpose. StreamIt contains basic constructs that expose the parallelism and communication of streaming applications without depending on the topology or granularity of the underlying architecture. The system structure is similar to ours since the program is specified as a graph of elements with dependencies. Nevertheless the system is built, as the name suggests, for streaming purposes, so its applicability is limited.

Intel (TBB) is the closest in spirit to our system. However, it does not support explicit construction of task graphs (graphs are constructed recursively via `spawn` call) and allows no customization of the Job Manager. TBB is limited to expressing dataflow dependencies, performs no architecture-specific optimizations and offers no special support for heterogeneous hardware.

6. CONCLUSION

We presented our preliminary work on Cascade, the parallel programming framework for complex systems. In the future we hope to use Cascade to address research challenges outlined in this paper and others.

Our experiences with DtC have indicated that our model of the simple acyclic graph is not yet rich enough to describe the complex interaction we wish to manage. The dependencies are currently absolute in the sense that nothing of the child will be executed until the parent is completely finished. We believe that a more robust system would allow further specification of types and parameters of the dependencies. We believe that a further increase in parallelism could be achieved if the architecture had facilities to allow a child to begin the computations when a predetermined portion of the parent workload has been completed.

Additionally we are aware that Cascade has only begun to address the intricacies of real-time dependencies and constraints. Future versions will dramatically increase the number of architectural constructs available to address these issues.

As has been mentioned, video games as prime example of complex soft real-time systems have their own specific needs distinct from other similar domains. Cascade currently uses a rather simple scheduling mechanism that only examines the local context (i.e. a nodes direct children) to determine scheduling order. To be fully suitable for the video game domain these will need to be addressed. A more complex mechanism is needed which is more aware of the entire tree and of possible heterogeneous processing units such as the SPU processors on the Playstation 3.

We would like to thank all the anonymous reviewers for their valuable observations and Tim Reid for his help in the early development of these ideas.

7. REFERENCES

[1] Code Demo: Destroy The Castle `http://softwarecommunity.intel.com/articles/eng/1363.htm`.

[2] The OpenMP specification for parallel programming `http://www.openmp.org`.

[3] A. Ghuloum, T. Smith, G. Wu, X. Zhou, J. Fang, P. Guo, B. So, M. Rajagopalan, Y. Chen, B. Chen. Future-Proof Data Parallel Algorithms and Software on Intel Multi-Core Architecture. *Intel Technology Journal*, 2007.

[4] M. Becchi and P. Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. *Proceedings of the International Conference on Computing Frontiers*, 2006.

[5] G. Blelloch and G. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Proceedings of the 2nd Symposium of Frontiers of Massively Parallel Computation*, pages 575–585, 1988.

[6] M. M. T. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow. Data parallel haskell: a status report. pages 10–18, 2007.

[7] A. Fedorova. Operating System Scheduling for Chip Multithreaded Processors. *Doctorate dissertation, Harvard University, Division of*

Engineering and Applied, 2006.

[8] M. Frigo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, 1998.

[9] M. Gordon, D. Maze, S. Amarasinghe, W. Thies, M. Karczmarek, J. Lin, A. Meli, A. Lamb, C. Leger, J. Wong, et al. A stream compiler for communication-exposed architectures. *ACM SIGARCH Computer Architecture News*, 30(5):291–303, 2002.

[10] K. Iverson. *A programming language*. John Wiley & Sons, Inc. New York, NY, USA, 1962.

[11] S. G. Jeffrey Dean. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[12] S. P. Jones. *Beautiful Concurrency*. O'Reilly Media, Inc., 2007.

[13] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic parallelism benefits from data partitioning. *SIGARCH Comput. Archit. News*, 36(1):233–243, 2008.

[14] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. pages 211–222, 2007.

[15] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. pages 287–296, 2008.

[16] M. McCool and R. Inc. Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform. *Proceedings of GSPx Multicore Applications Conference*, 2006.

[17] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[18] A. Snavely and D. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[19] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. pages 2–13, 1998.