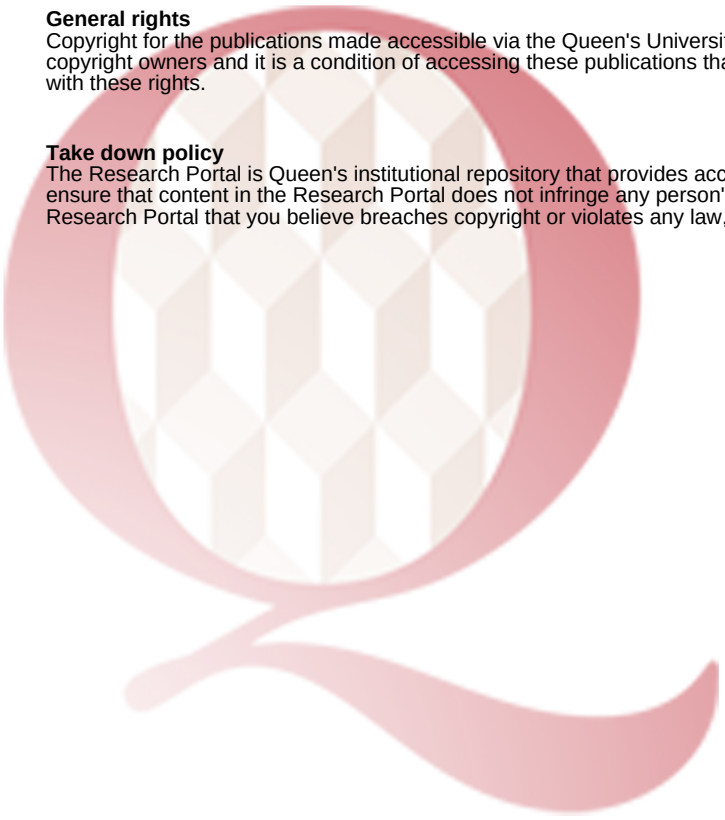
# Parallel Programming of General-Purpose Programs Using Task-Based Programming Models





# Parallel Programming of General-Purpose Programs Using Task-Based Programming Models

Hans Vandierendonck*, Polyvios Pratikakis† and Dimitrios S. Nikolopoulos†
* Dept. of Electronics and Information Systems, Ghent University, Ghent, Belgium, Email: hvdieren@elis.ugent.be
†Institute of Computer Science, Foundation for Research and Technology – Hellas (FORTH)
Heraklion, Crete, Greece, Email: {polyvios,dsn}@ics.forth.gr

***Abstract*—The prevalence of multicore processors is bound to drive most kinds of software development towards parallel programming. To limit the difficulty and overhead of parallel software design and maintenance, it is crucial that parallel programming models allow an easy-to-understand, concise and dense representation of parallelism.**
**Parallel programming models such as Cilk++ and Intel TBBs attempt to offer a better, higher-level abstraction for parallel programming than threads and locking synchronization. It is not straightforward, however, to express all patterns of parallelism in these models. Pipelines are an important parallel construct, although difficult to express in Cilk and TBBs in a straightforward way, not without a verbose restructuring of the code.**
**In this paper we demonstrate that pipeline parallelism can be easily and concisely expressed in a Cilk-like language, which we extend with input, output and input/output dependency types on procedure arguments, enforced at runtime by the scheduler. We evaluate our implementation on real applications and show that our Cilk-like scheduler, extended to track and enforce these dependencies has performance comparable to Cilk++.**

## I. Introduction

The onset of multicore processors has brought a growth of novel parallel programming models aiming to facilitate multicore programming. The currently dominant parallel programming paradigm of threads and lock-based synchronization requires the programmer to reason about the myriad implicit and explicit interactions of threads through shared memory and synchronization all by hand, making parallel programming difficult and error prone.[1]

There have been several attempts to replace threads and lock-based synchronization with better abstractions. For instance, OpenMP uses the abstraction of a *task*, a basic unit of parallel work, as an alternative abstraction to threads. Extending this idea, Cilk marks recursive, parallel tasks in a program and a scheduler optimizes the concurrent execution of tasks at runtime. This abstraction is also advocated by Intel's Threading Building Blocks [1], Microsoft's Parallel Pattern Library and SMPSS [2], amongst others.

The Cilk abstraction of parallel recursive tasks can easily describe some parallel patterns, such as DOALL loops or divide-and-conquer parallelism. However, it is not as easy to describe pipeline parallelism, parallel-stage pipelines and speculative parallelism using nested parallel tasks. Yet, these constructs occur very frequently in general-purpose programs [3], [4].

This paper extends the Cilk programming model to greatly increase the readability and density of programming such parallel structures. We augment the Cilk model of parallel execution by adding dependency clauses on task arguments. Argument dependency types indicate that an argument is an *input* to the task, an *output* or both (*inout*). We show that our system facilitates programming complex patterns of parallelism like parallel pipelines compared to Cilk, without loss of performance. We also postulate that it facilitates exploiting speculative parallelism. Overall, we believe this extension benefits *ease-of-programming*, code density and code readability for general-purpose parallel programming.

In the remainder of this paper, we first discuss related programming models (Section II) and introduce our task-based programming model with nested tasks (Section III). Next, we describe the two extensions to the Cilk language and runtime that are required by our model: versioned objects, a new type of hyperobject [5], that automatically manages renaming of objects in order to increase the amount of parallelism (Section IV) and an extension to the Cilk scheduler to handle task dependencies (Section V). We demonstrate that pipelining constructs can be easily represented in our language without impacting execution time (Section VI). Furthermore, we discuss how the model could facilitate writing speculatively parallel programs in Section VII. Section VIII concludes this paper.

## II. Related Work

Libraries for parallel programming hide the complexity of parallel programming from the user of the library by providing a set of functionalities that orchestrate the concurrent execution of parallel programs. Several such libraries exist or are under continued development, e.g. Intel's Threading Building Blocks [1], STAPL [6] and Galois [7]. These libraries, however, are often tailored to specific problem domains, e.g. STAPL provides a thread-safe standard template library while Galois facilitates optimistic execution of amorphous programs.

StreamIt [8] is a language that targets streaming computations. StreamIt makes heavy use of pipeline parallelism, although it is a domain-specific language and as such too restrictive and not fit for general-purpose parallel programming. StreamIt differs significantly from our programming model in

[1]At the moment this text is written, a Google query for the phrases "parallel programming" and "difficult" produces more than 27,000 technical papers, 447 of which use the phrase "notoriously difficult."

the sense that StreamIt heavily depends on the notion that a task is repeatedly applied to a stream of data. Tasks can place or get multiple data items on a stream. As such, a program is represented statically as a set of tasks that are linked by streams. The main challenge is to compile the task invocations and data communications efficiently. In contrast, in our work a directed acyclic graph (DAG) of dependent task instances is constructed dynamically. There is no notion of streams, only of dependencies between task instances. A dynamic scheduler decides when to execute which task instance.

The Cilk [9]/Cilk++ language is one of the best known task-based programming models. Parallelism is expressed using a *spawn task* statement while a *sync* statement forces a parent task to wait until all its children are finished. Tasks may be nested up to arbitrary depth. An important component of Cilk is its scheduler, which determines at runtime what tasks are executed concurrently. Cilk implements a distributed work-stealing scheduler for shared memory systems. Cilk programs have been shown to take space and time optimal to within a linear factor of the *C elision* of the program [10], the C code obtained after stripping Cilk keywords.

In order to further facilitate parallel programming, Cilk++ provides *hyperobjects* [5]. Hyperobjects offer each thread a private view of a shared C/C++ object, allowing threads to execute concurrently without races. Three types of hyperobjects have been identified: reducers, holders and splitters [5]. The runtime system guarantees that properly defined reducers compute exactly the same value as would be computed by the serialization of the program.

Argument dependencies have already been shown to improve performance in cases of irregular dependency patterns. SMPSS [2] is a task-based programming model with a focus on scientific computations. It has its roots in the CellSS model for executing programs on the Cell B.E. processor [11]. In SMPSS, a single master thread executes a program and collects all spawned tasks for parallel execution by a set of worker threads. Task arguments are labeled with a dependency type, either *input*, *output*, *inout* or *reduction*. These have the obvious meaning, that an argument is either read-only, write-only, read-write or that it is part of a reduction operation. The dependency types allow the runtime system to schedule tasks on worker threads in much the same way as a superscalar processor schedules instructions on its execution units.

SMPSS does not admit nested parallelism. Instead, it extracts parallelism from analyzing the dependency graph of a large set of tasks. Furthermore, the runtime system performs renaming (privatization) of task arguments in order to break anti-dependencies and output dependencies. Breaking those dependencies increases task parallelism.

## III. Nested Task Model with Task Dependencies

Our nested task model supports nested task creation as in Cilk++ and dependency types on task arguments as in SMPSS. We support input, output, and inout dependencies. We support reducer hyperobjects instead of reduction dependencies as reducers will introduce less contention than the lock-based approach of SMPSS.

```
 1 int  stage_A(outdep<int[]> ab, inoutdep<struct S> a);
 2 void stage_B(indep<int[]> ab, outdep<float[]> bc);
 3 void stage_C(indep<float[]> bc, inoutdep<struct S> c);
 4
 5 void a_pipeline() {
 6     versioned<struct S> a, c;
 7     versioned<int[100]> ab;
 8     versioned<float[200]> bc;
 9     while( 1 ) {
10         if (stage_A(ab, a))
11             break;
12         spawn stage_B(ab, bc);
13         spawn stage_C(bc, c);
14     }
15     sync;
16 }
```

Fig. 1. A program containing pipeline parallelism expressed in the proposed task-based programming model. We assume a Cilk-like syntax, extended with dependency types on function arguments. The A-stage returns the loop termination condition as an integer value, but the same approach works for counted loops.

If a task does not have any arguments labeled with a dependency type, then the task spawn is *independent*, and corresponds to a Cilk spawn: it is always valid to execute the spawned task. If the task signature lists at least one argument with a dependency type, then the task spawn is *dependent*, and may not be executed immediately. If the task dependencies are not satisfied (i.e. an object listed in an input or inout dependency has not been computed by another task) then we postpone the execution of the task to a later point. We adapt the scheduler to reconsider such postponed tasks later in the program execution, e.g. at a sync statement.

Note that not all task arguments must be annotated. It is assumed that task arguments that are not annotated simply do not give rise to potential memory dependencies with concurrently executing tasks.

Similarly, sync statements may now also be independent or dependent. Independent sync statements take no arguments and suspend the syncing task until all children tasks have terminated, as in Cilk. Dependent sync statements take a set of objects as arguments, and suspend the task until all the arguments have been computed. This allows the task to access objects computed by some of its children without restricting parallelism of all children with an independent sync.

Tracking and enforcing task dependencies is a crucial part of the system. Hereto, it is necessary to determine at runtime the status (at least a busy/ready flag) of objects passed to task arguments with a dependency type. In this work, we introduce *versioned objects* to allow efficient and automatic dependency resolution and renaming (privatization) of objects.

Figure 1 shows an example program using our programming model. We extend Cilk with dependency types on task arguments. Only versioned objects may be passed to such arguments, indicated by the *versioned* label. This label attaches metadata to the object, allowing the runtime system to track producer-consumer relations and to create multiple versions of these objects during execution. Note that the dependency

types completely capture the nature of the pipeline: the programmer simply declares the inputs and outputs of each task implementing a stage of the pipeline, and the scheduler will enforce the correct order on all tasks.

Expressing the same program using Intel TBB takes 36 lines of code and requires the programmer to manually manage renaming. We have also devised a Cilk++ solution using a reducer hyperobject to enforce the serial execution of the last pipeline stage. The Cilk++ version of the code takes 34 lines while the (reusable) hyperobject takes about 130 lines more. Again, the programmer must manage renaming manually. We have also investigated using Microsoft's ConcRT to implement parallel-stage pipelines using the Asynchronous Agent Library. Our conclusion is that this is probably possible, but far from evident because the programmer must control replication of the B-stage actor to extract parallelism in this stage and she must also manually enforce reordering of tasks before executing them on the final serial C-stage. Renaming is also a responsibility of the programmer. Overall, our model allows a much denser and readable specification of the parallelism compared to existing approaches.

To increase confidence in our system, we have formalized our dependency resolution algorithm to show that it produces execution orders equivalent to the sequential execution of the program. Namely, we defined both the sequential and the parallel operational semantics of a simple functional language with Cilk-like recursive parallelism and dependency-aware scheduling. Our sequential operational semantics treat `spawns` as simple function calls. We defined the property of *sequential equivalence* to be the equivalence of any terminating parallel execution to a sequential execution that produces the same result, and showed that such a sequential execution always exists for our dependency-aware scheduling. Note that the property of sequential equivalence is stronger than the serializability property that holds for Cilk.

Intuitively, to prove that the dependency-aware scheduling satisfies sequential equivalence, we show how to construct the trace of a sequential execution that computes the same result as any given parallel execution trace. We define the reorderable operations in the parallel trace as those that can be reordered without changing the end result of the computation. The proof is by induction on the execution trace, where we split the parallel execution trace into a sequential trace followed by a parallel trace, and inductively grow the sequential execution trace by reordering operations in the parallel execution trace. In short, we show that any parallel interleaving is equivalent to the interleaving where the first step of the execution follows the sequential order, and generalize by induction.

## IV. Versioned Hyperobjects

Versioned hyperobjects implement two functionalities of the runtime system: (i) tracking task argument dependencies and (ii) versioning objects (a.k.a. renaming or privatization) when it is helpful to increase parallelism. Versioned objects are a type of hyperobject as they automatically provide distinct views to threads that reference the same variable.

### A. Dependency Tracking

The runtime system tracks and enforces argument dependencies by monitoring whether the objects passed as arguments are ready. This metadata is associated to individual views or versions of the objects. The dependency tracking metadata consists of a readers head and tail pointer and a writers head and tail pointer. The head and tail pointers function as if they were the head and tail pointers of a FIFO. The tail pointers are incremented when trying to spawn a task. The head pointers are incremented when completing a task.

The counters allow implementing the basic tests necessary to determine when a task is ready to execute. For instance, *inout* dependencies must wait until all prior readers and writers have finished. This is implemented by assigning to the inout task two tag values that are stored with the task in the pending task list. The tag values are set equal to the tail pointers as they are encountered in serial order (when the spawn is attempted). When the last completing writer updates the head pointer, the head becomes equal to the write tag and the required version is available. Similarly, when the last prior reader completes, the reader head becomes equal to the read tag and the inout task becomes the sole reader/writer.

The use of a FIFO-like setup does not mean that all tasks are executed serially. For instance, if multiple tasks with an input dependency are registered for a view, all of them can issue as soon as the writer head equals their writer tag (the reader tag plays no role here).

### B. Versioning

Versioning is directed by the need to break anti dependencies and output dependencies. As such, versioning occurs only when spawning tasks with an *output* dependency. And this happens *only* if prior readers or writers are pending. Inout dependencies are never renamed in our implementation, although that might make sense in some situations, e.g. to avoid stalling an inout dependent task as long as there are readers (input dependency) active.

When a versioned object is passed to a spawned task with an input or inout dependency, then the runtime system arranges for the parent and child procedures to *share* the same view of the object. When the child returns, the parent's view is maintained. The child's view is destroyed if the child is the last user of that view.[2]

Output dependencies may give rise to renaming the object (creating a new view) when the object is in use by pending tasks at the time of a spawn. In this case, the child receives the parent's view (the existing version) and the parent receives a *new empty view*. Otherwise, parent and child procedures *share* the existing view. Views are joined as in the other cases.

## V. Task Scheduling

We implemented a Cilk-like work-first scheduler assuming a shared-memory model. Our motivation for this type of

[2]The metadata stored with a view includes a reference counter in order to allow recycling memory consumed by inaccessible versions.

scheduler is that it is well-suited to irregular and nested parallelism, as we expect to find in general-purpose programs.

In the absence of dependency type arguments, the scheduler behaves like the Cilk scheduler as described in [5], attempting serial execution of the inner loops and favoring work stealing at the outer loop level. When the arguments of a child task are not ready, then the task is added to a list of pending tasks in the parent's stack frame.

There are several design choices for when to retry execution of pending tasks. In our current implementation, pending ready tasks are retrieved when the scheduler tries to perform a provably-good-steal operation. A provably-good-steal occurs either when returning from a spawn that is (or has been) executing concurrently with the parent, or when a procedure waits at a sync statement for its children to terminate [5].

It has been shown that the Cilk scheduler incurs linear overhead of stack space usage and linear overhead in execution time [10]. It is clear that these properties are not maintained when extending the scheduler with the ability to postpone spawns, as the total number of constructed but unevaluated stack frames grows. The exact deviation of linear stack space growth however depends on the executed program, in particular on the number of postponed spawns.

## VI. Evaluation

We present a preliminary evaluation of our runtime system that implements task argument dependencies. The runtime system is essentially a Cilk clone written as a C++0x library. C++0x is very expressive, allowing us to encode code transformations that would otherwise require a source-to-source compiler. The library implements *spawn* and *sync* as function calls. It implements a cactus stack and executes spawned tasks on cactus stack frames in order to allow stealing parent stack frames.[3] Because of this, all function calls must be intercepted by means of a *call* function in order to setup the stack frame. For performance reasons, a *leaf_call* function is provided that executes unmodified code on the linear stack, but does not allow calling back to the parallel runtime.

This baseline runtime system is extended with object dependency tracking and enforcement as described in this paper. Versioned objects are implemented as template classes similarly to hyperobjects in Cilk++. Input, output and inout dependency types on task arguments are implemented by defining three more template classes. The runtime system recognizes task spawns with arguments of these types and automatically applies dependency tracking and enforcement.

For reference, we compare performance with Cilk++ version 8503, implemented in GCC 4.2.4. We use GCC 4.4.1 for our runtime because it requires C++0x support.

Note that we advocate here a use of task argument dependencies in order to simplify parallel programming in cases where the usual spawn and sync primitives are insufficiently expressive. What we show next is that this "ease of programming" does not carry a performance penalty. We parallelized two programs using object dependencies: bzip2 and hmmer.

[3]In contrast, Cilk stores only procedure-local data in the cactus stack frames and executes the tasks on the scheduler thread's linear stack.

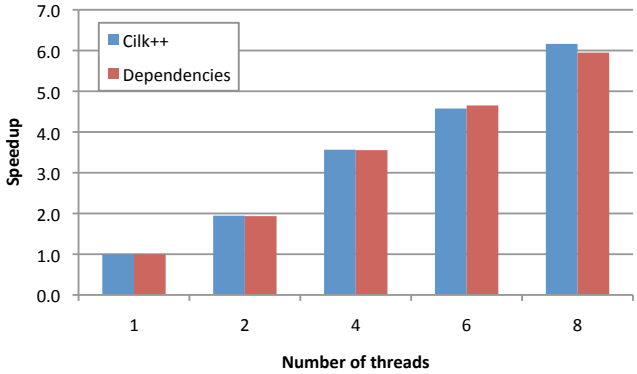



Fig. 2. Speedup results of bzip2 compression on the Cilk++ version and the version using task argument dependency types.

### A. Bzip2

A parallel version of bzip2 compression has been created by Cilk Arts [12]. Bzip2 compression has a parallel-stage pipeline where a first serial stage reads a block of data from a file, the second parallel stage compresses the data and the third serial stage writes the compressed data to an output file.

The Cilk++ version uses a reducer hyperobject to orchestrate writing to the output file. The reducer basically queues up all output data in memory until all logically preceding data has been written to file. This approach has the downside that data is written twice: once to an in-memory buffer and a second time when copying the buffer to the file. Note that copying the buffer may require bit-shifting because blocks do not necessarily start on a byte boundary.

When using object dependencies, the function call to write to the output file is moved from the compression function to a spawn statement in the main compression loop. By inserting the necessary object dependencies, the runtime system ensures that all calls to the writer function occur in correct order.

Figure 2 summarizes the speedups observed using the Cilk++ code and the code using object dependencies. Results are reported for strong ('-9') compression of a badly compressable 250MB file, as suggested by [12]. We also read blocks before applying run-length encoding [12]. These measurements indicate that essentially the same performance is obtained by using task argument dependencies as when using reducers.

### B. Hmmer

The main loop in the calibration code of hmmer contains a three-stage pipeline where the first serial stage generates a random sequence, the second parallel stage evaluates a Hidden Markov Model (HMM) on this sequence and the third stage accumulates characteristics of the HMM in a histogram. The first stage is best modeled as serial in order to be able to reproduce random numbers. The third stage actually only requires exclusive access to the histogram, but we show that

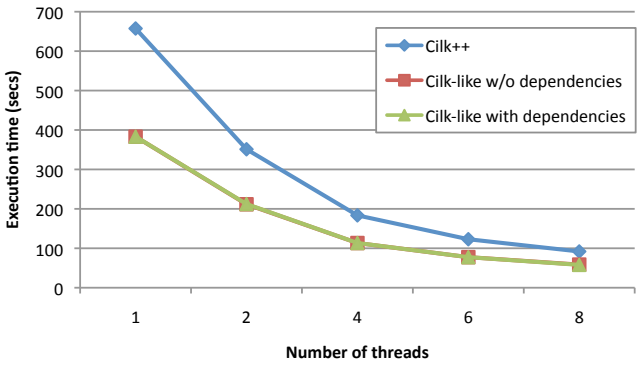



Fig. 3. Execution time of hmmer on the Cilk++ version and our runtime with and without task argument dependency types.

enforcing serial ordering on it does not harm performance. Exclusive access is obtained using a compare-and-swap lock.

Figure 3 shows the execution time on a SPEC reference input obtained with Cilk++, our Cilk-like runtime without using argument dependencies and our runtime with argument dependencies. First note the big performance difference on single-threaded code. This is due to differences in GCC versions used by Cilk++ and our runtime.

The results (Figure 3) show that our runtime obtains similar scalability as Cilk++. Furthermore, adding object dependencies to the model hardly impacts performance.

Note that both test programs perform hundreds of millions of instructions worth of work per spawn. As such, stealing is infrequent and the proposed extension to the Cilk scheduler is sufficient. In future work, we will study the scheduler design also for fine-grain tasks.

## VII. Potential for Speculative Execution

Speculative execution is an important source of parallelism [3], [13]–[15]. We describe how task-based programming models may facilitate speculative execution. This is not implemented.

Figure 4 shows a code mock-up of a task-based speculative program. The speculative parallelism in this example matches for instance the vpr program which has been discussed in the literature [3]. Speculative parallelism exists in phases of the algorithm where the *accept()* is rarely executed. It can be exploited by respecifying the input dependence of the state argument of *try_swap()* as a speculative input dependence (specindep). This indicates the speculative nature of the spawned task but also implies that dependencies with pending outdeps may be ignored. For instance, *try_swap()* is not considered dependent on prior instances of *accept()* because it speculates on the shared *state* argument.

When *accept()* executes, the *abort* statement indicates that all computations recursively dependent on the current versions of the variables *state* and *i* should be re-executed.

Note that Cilk (but not Cilk++) also allows speculative execution of this sort using inlets and the abort statement. Task-based programming models are however more generic because they make it easy to mix other constructs in the same loop. E.g. a third task could be added to the loop that

```
1  void try_swap(indep<struct S> state,
2                outdep<int> success,
3                outdep<struct C> changes);
4  void  accept(inoutdep<struct S> state,
5               indep<struct C> changes);
6
7  void speculated() {
8      versioned<struct S> state;
9      versioned<struct C> changes;
10     versioned<int> success;
11     versioned<int> i;
12
13     for( i=0; i < N; ++i ) {
14         spawn try_swap((specindep<struct S>)state,
15                        success, changes);
16         spawn {
17             if ( success ) {
18                 abort state, i;
19                 accept(state, changes);
20             }
21         }
22     }
23     sync;
24 }
```

Fig. 4. Code mock-up for speculative parallelization.

uses the modified state non-speculatively. A task-based model executes this third task only non-speculatively and overlaps its execution with the remainder of the loop. Non-speculative tasks would never be aborted, because the runtime knows what tasks are speculative. To the best of our knowledge there is no equivalent construction in Cilk.

## VIII. Conclusion

This paper advocates the use of task-based programming models with nested task spawning for writing general-purpose programs. Such programming models simplify the specification of irregular parallelism. Programming constructs that benefit include pipelines, pipelines with parallel stages and non-linear pipelines.

Our programming model greatly enhances the ease of programming such constructs by (i) expressing the parallelism densely and (ii) automatically renaming or privatizing objects when this improves performance.

We extended a Cilk-like scheduler to recognize and enforce argument dependency types on task spawns. Preliminary evaluation shows that the ease-of-programming can be achieved without loss of performance on bzip2 compression and Hidden Markov Model evaluation.

## Acknowledgments

Hans Vandierendonck is a Postdoctoral Fellow of the Research Foundation – Flanders (FWO). He is sponsored by a Travel Grant of the FWO. The research leading to these results has received funding from the European Community's 7th Framework Programme [FP7/2007-2013] under the ENCORE Project (http://www.encore-project.eu), grant agreement no. 248647, the TEXT Project (http://www.project-text.eu/), grant agreement no. 261580, and under the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC, http://www.hipeac.net), grant agreement no. 217068.

## REFERENCES


[1] *Intel Threading Building Blocks*, Intel, Sep. 2010, document Number 319872-006US.

[2] J. M. Perez, R. M. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multicore architectures," in *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2008, pp. 142–151.

[3] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August, "Revisiting the sequential programming model for multi-core," in *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 69–84.

[4] A. Kejariwal, X. Tian, W. Li, M. Girkar, S. Kozhukhov, H. Saito, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos, "On the performance potential of different types of speculative thread-level parallelism," in *Proceedings of the 20th annual international conference on Supercomputing*, 2006, pp. 24–.

[5] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin, "Reducers and other Cilk++ hyperobjects," in *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, 2009, pp. 79–90.

[6] L. Rauchwerger, F. Arzu, and K. Ouchi, "Standard templates adaptive parallel library (STAPL)," in *LCR '98: Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, 1998, pp. 402–409.

[7] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, 2007, pp. 211–222.

[8] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, "A stream compiler for communication-exposed architectures," in *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, 2002, pp. 291–303.

[9] M. Frigo, C. E. Leierson, and K. H. Randall, "The implementation of the Cilk-5 multi-threaded language," in *PLDI '98: Proceedings of the 1998 ACM SIGPLAN conference on Programming language design and implementation*, 1998, pp. 212–223.

[10] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*. 1994, pp. 356–368.

[11] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: A programming model for the Cell BE architecture," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[12] J. Carr, "A parallel bzip2," http://software.intel.com/en-us/articles/a-parallel-bzip2/, Apr. 09, retrieved Jan. 19th, 2011.

[13] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August, "Speculative decoupled software pipelining," in *Parallel Architectures and Compilation Techniques, International Conference on*, 2007, pp. 49–59.

[14] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang, "Software behavior oriented parallelization," in *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, 2007, pp. 223–234.

[15] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," in *Proceedings of the 27th annual international symposium on Computer architecture*, 2000, pp. 1–12.