Erbium: A Deterministic, Concurrent Intermediate Representation for Portable and Scalable Performance

Cupertino Miranda[1], Philippe Dumont[1,2], Albert Cohen[1],
Marc Duranton[2] and Antoniu Pop[3]

[1] INRIA Saclay and LRI, Paris-Sud 11 University, France

[2] NXP Semiconductors, The Netherlands

[3] Centre de Recherche en Informatique, MINES ParisTech, France

December 14, 2009

Abstract

Tuning applications for multi-core systems involve subtle concepts and target-dependent optimizations. New languages are being designed to express concurrency and locality without reference to a particular architecture. But compiling such abstractions into efficient code requires a portable, intermediate representation: this is essential for modular composition (separate compilation), for optimization frameworks independent of the source language, and for just-in-time compilation of bytecode languages. An intermediate representation is also essential to library and other baseline computing infrastructure developers who cannot afford the abstraction penalty of higher-level languages. But efficiency is nothing if it ruins productivity of the few available experts. Efficiency programmers need an alternative to fragile, ad-hoc optimizations built upon non-deterministic primitives. This paper introduces Erbium, an intermediate representation for compilers, a low-level language for efficiency programmers, and a lightweight runtime implementation. It is built upon a new data structure for scalable and deterministic concurrency, called *Event Recording* (Er). Our work is inspired by the semantics of data-flow and synchronous languages, motivated by advanced optimizations relying on non-blocking concurrency. We provide experimental evidence of the productivity, scalability and efficiency advantages of Erbium, relying on a prototype implementation in GCC 4.3.

1 Introduction

To cope with the ever increasing demand for performance, hardware engineers design multi-core processors and accelerators. The clock race is over: increasing performance requires changing the code structure to harness complex parallel hardware and memory hierarchies. But this is a nightmare for programmers: translating more processing units into effective performance gains involves a never-ending combination of target-specific optimizations. These optimizations involve subtle concurrency concepts, often non-deterministic algorithms, as well as target-dependent enhancements of memory locality. Optimizing compilers and runtime libraries do not shield programmers from the complexity of the hardware; as a result the need for manual, target-specific optimizations increases with every processor generation.

Higher-level languages are designed to express (in)dependence and locality without reference to any particular hardware, leaving compilers and runtime systems with the responsibility of lowering these abstractions to well-orchestrated threads and memory management. The current practice tends to isolate the (sequential) language from the threading libraries. It induces severe inefficiencies, validation and verification nightmares, and ultimately, undetected non-deterministic bugs. This paper introduces ERBIUM (`Er`) an intermediate representation for compilers, a data structure for scalable and deterministic concurrency, and a lightweight runtime implementation. ERBIUM is a deterministic, portable abstraction, to expose data-level, task and pipeline parallelism suitable to a given target. It is meant as an intermediate representation for compilers, for active/autotuned library generators, and for efficiency programmers.

Our experiments on a real application demonstrate a speedup of 10.1 on a 24-core Xeon and 9.51 on a 16-core Opteron, from thread-level parallelization alone (GCC `-O2`). Combined with more aggressive optimizations, including automatic vectorization (GCC `-O3`), the speedups climb to 12.6 and 14.6, respectively. This peaceful collaboration of thread-level parallelism with middle- or back-end optimizations is not customary with high-level languages or with low-level library-based threading approaches.

The rest of the paper is structured as follows. Section 2 discusses the design goals and choices of the intermediate representation. Section 3 defines its syntax and semantics. Section 4 details its transparent specialization on shared-memory platforms. Section 5 evaluates our implementation on realistic and extreme situations. Section 6 discusses related work. We conclude and summarize important research directions in Section 7.

2 Design of the Intermediate Representation

ERBIUM defines an intermediate representation for compilers and usable as a low-level language by efficiency programmers. We aim at the simultaneous satisfaction of the following objectives.

Determinism. ERBIUM's semantics derives from *Kahn Process Networks* (KPNs) [26]. KPNs are canonical concurrent extensions of (sequential) recursive functions preserving *determinism* (time independence) and *functional composition*. Functions in a KPN operate on infinite data *streams* and follow the *Kahn principle*: in denotational semantics, they must be continuous over the Scott topology induced by the prefix ordering of streams [26,31]. A classical operational definition of KPNs states that processes communicate through lossless FIFO channels with blocking reads and non-blocking writes. The semantics of ERBIUM is not bound to this particular operational implementation. ERBIUM processes can be arbitrary, imperative C code, operating on *process-private data* only; their interactions are compatible with the Kahn principle, with an operational semantics favoring scalable and lightweight implementation.

Modularity. Separate compilation of modular processes is essential to the construction of real world systems. An ERBIUM program is built of a sequential main thread spawning interacting processes dynamically. Modularity has triggered much research in high-level concurrent languages, regarding causality, liveness [8,32], and resource boundedness [9–11,22]. Our low-level design is complementary to these approaches: we rest on the guarantees of the compiler front-end in charge of lowering high-level abstractions to ERBIUM. But resource management remains a task of the generated code and runtime library: we introduce a low-level mechanism for modular back-pressure supporting wide broadcast and worksharing scenarios for large-scale data-parallelism.

Expressiveness. In concurrent data-flow languages, data and functional parallelism is implicit in (recursive) functions [27,39]. As an intermediate representation, ERBIUM provides explicit, asynchronous spawn points for concurrent processes.

A runtime system relying on context-switches — even lightweight user-space ones — cannot match the speed of hardware synchronization. To hide communication latency and avoid such overhead, ERBIUM favors long-running processes and data-flow communications. To this end, traditional data-flow models implement data streams with `push()`/`pop()` primitives over FIFO channels. ERBIUM's data structure for communication is much richer, allowing for *random-access peek* (use), *poke* (definition, assignment) and communications decoupled from the actual synchronization. This data structure is called an *event recording* (Er), or *recording* for short. It unifies the *stream* and *future* [23] concepts, and generalizes them to support single-producer/multiple-consumer concurrency.

Unlike periodic subclasses of KPN [6,11,30,49], ERBIUM supports dynamic creation, termination of concurrent processes, allowing for arbitrary mode switches, resets and adaptation scenarios. Determinism is preserved through generic initialization and termination protocols.

Static adaptation. As an intermediate language, ERBIUM supports aggressive specialization, analysis and optimization. It is not restricted to periodic subclasses of KPN [20,28] or specific parallel computation skeleton [16,43]. It supports program transformations for dynamic, data-dependent control flow applications, including generalized forms of decoupled software pipelining [21,37,42].

Binary code does not offer the required level of static adaptation; on the contrary ERBIUM *defines the intermediate representation as the portability layer*. ERBIUM is accessible to efficiency programmers as a source language and serves as a target for higher-level languages and compilers. This is an essential design decision. The ERBIUM runtime may be specialized for different memory models, shared (globally addressable, cache coherent) or distributed. It may transparently exploit any hardware acceleration for faster context switch, synchronization and communication.

The compiler is responsible for selecting an appropriate specialization, offering the most relevant run-

time primitives and interface for a given platform. It also adapts the grain of concurrency, implementing fusion of task pipelines, static scheduling of task graphs, coarsening of the communication/synchronization grain, and automatic buffer size inference. Exploiting split-phase synchronization and communication at compilation time allows to overlap communications with computations.

Lightweight implementation. ERBIUM is designed to be as close as possible to the hardware while preserving portability and determinism. This is a central motivation for a low-level abstraction. Any intrinsic overhead in its design and any implementation overhead will hit scalability and performance; such overheads cannot be recovered by a programmer who operate at this or higher levels of abstraction.

Thanks to its data-flow semantics, it is possible to implement the concurrent primitives of the ERBIUM runtime only relying on *non-blocking synchronizations*. No busy waiting and no system call is involved — apart from the necessary stalls to implement (back-)pressure. Thanks to its native support, *broadcast and worksharing patterns are very efficiently implemented*, avoiding unnecessary copy in (collective) scatter operations.

Finally, ERBIUM can be implemented with a very low memory footprint thanks to its low-level, streamlined design. Code footprint is only 59 *kB* on x86-64 (compiled with GCC 4.3 `-O3`); more importantly, on a distributed-memory IBM Cell, it is broken down into 28 *kB* for the PPU (PowerPC) and 11 *kB* for each SPU (accelerator). The data footprint is minimal, with small recording descriptors and few shared variables (aside from communication buffers whose size depend on the application and on fine-tuning to the target platform).

3 Semantics of the Intermediate Representation

Formalization is out of the scope of this practice- and design-oriented paper; we use a C syntax and informal semantics instead.

Figure 1 illustrates the ERBIUM primitives and event recording structures on a simple producer-consumer template. We will use this example to define data-flow-synchronization and communication, resource management, process creation and termination.

Data-flow synchronization and communication. An *Event Recording* (`Er`), or *recording* for short, is an unbounded stream, indexed in the set of natural integers and randomly addressable. Each recording is associated with a private, monotonically increasing *commit* index. Recording elements at indices *less than the commit index* are fully defined and *read-only*.

A *view* is a *read-only* unbounded stream, randomly addressable, connected to a recording. Each view is associated with a private, monotonically increasing *update* index. View elements at indices *less than*

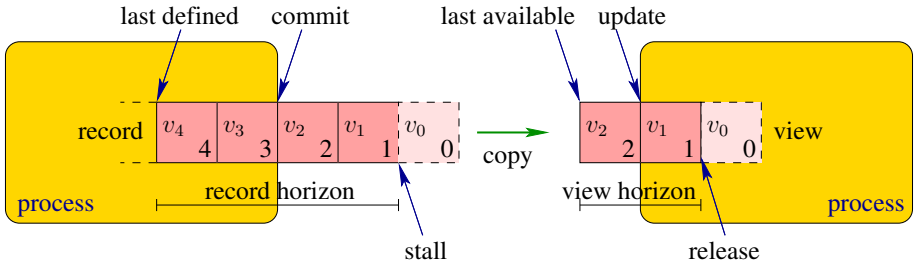

Figure 1: Producer-consumer data flow with bounded resources

```
int main() {
  recording int re =
    new_recording(1);
  run producer(re);
  run consumer(re);
}
```

```
process producer
  (recording int re) {
  int tl=0, hd, i;
  alloc(re, P_HORIZ);
  while (1) {
    hd = tl + P_BURST;
    if (hd<N) break;
    stall(re, hd);
    for (i=tl; i<hd; i++)
      re[[i]] = foo(i);
    commit(re, hd);
  }
}
```

```
process consumer
  (recording int re) {
  int tl=0, hd, i;
  int sum=0;
  view int vi = new_view(re);
  register(vi);
  alloc(vi, C_HORIZ);
  while(1) {
    hd = tl + C_BURST;
    receive(vi, hd);
    hd = update(vi, hd);
    if (!hd) break;
    for (i=tl; i<hd; i++)
      sum += vi[[i]];
    release(vi, hd);
    tl = hd;
  }
}
```

Figure 2: Producer-consumer example

the update index are identical to the the corresponding elements of the connected recording. `recording T r` (resp. `view T v`) declares a recording `r` (resp. view `v`) of data elements of type `T`. The `[[i]]` syntax is used to subscript a recording or view at index `i`.

On Figure 1, the producer process committed indices 0, 1 and 2 to the recording, but keeps indices 3 and 4 private. At this point, modifications of these values are still possible on indices 3 and 4, but the values at indices 0, 1 and 2 are read-only. On the consumer side, only indices 0 and 1 have been updated into the view; index 2 is available but the consumer did not yet decide to observe it in the view. All three values are read-only.

The `commit()` and `update()` primitives implement data-flow *pressure*, enforcing causality among processes. `void commit(recording T r, int i)` increments the commit index of `r` to `i`; it does nothing if `i` is lower than or equal to 0 or to the current commit index. `int update(view T v, int i)` sets the update index of `v` to `i`, or does nothing if `i` is lower than or equal to 0 or to the current update index. It waits until the commit index of the connected recording is greater than or equal to `i`. It returns the value of `i`, except on termination of process owning the connected recording, to be explained in the termination section. This primitive controls what indices will be observed in the process controlling the view; it offers the same determinism guarantees as a blocking read in a KPN.

Indices are always non-negative integers. The update index of a view is always less than or equal to the commit index of its connected recording. On the other hand, the *observed* value of the commit index (not directly accessible to the program) depends on the observing thread and on how frequently its is synchronized across the machine; we only assume that changes to the commit index will *ultimately* be observable by any hardware thread.

In our split-phase design, data-flow communication is decoupled from synchronization. A one-sided, asynchronous communication is initiated with the `receive()` primitive. `void receive(view T v, int i)` updates `v` with the data of its connected recording, starting from the current update index up to `i`-1. It cannot complete until the commit index reaches `i`. The communication may also proceed earlier, retrieving sub-ranges of indices, this is left to the implementation and not observable at the level of the intermediate representation. When an asynchronous call to `receive(v, i)` is pending, a followup `update(v, j)` must wait until all elements of indices lower than $\min(\texttt{i}, \texttt{j})$ have been retrieved. On shared-memory platforms, `receive()` may be implemented as prefetch or no-op. On distributed-memory platforms, asynchronous communication will typically involve a DMA circuit.

On Figure 1, indices 0, 1 and 2 have been received by the consumer and stored in the view's buffer. This is independent from the fact that the consumer did not yet decide to observe index 2.

Resource management. Practical implementation of recordings need a bounded memory space: reasonable KPNs can be evaluated with bounded memory.[1] The bound may be managed statically or dynamically, and corresponds to the maximal number of live elements. The live elements of a recording (resp. view) are stored in a *sliding window* and its size is called the recording's (resp. view's) *horizon*. When compiling for shared memory platforms, sliding windows of the recording and views may be coalesced to avoid unnecessary data copying.

Some primitives are required to stop the production of data that would violate the semantics of an unbounded recording, while operating on a finite horizon. As with any blocking write semantics, these primitives may induce resource deadlocks when the recording horizon is insufficient. In general, it is the responsibility of compiler front-end and/or of the runtime library, to provide algorithms for automatic inference of buffer sizes (static and/or dynamic) [10,11,13,22,30].

The `release()` and `stall()` primitives implement *back-pressure*. An element is considered live in a view as long as `release()` has not been called on a higher index. An element is considered live in a recording as long as at least one connected view has not yet released its index, calling `release()` on a higher index. Each recording (resp. view) is associated with a private, monotonically increasing *stall* (resp. *release*) index, marking the tail of live elements in the recording (resp. view). `void stall(recording T r, int i)` waits as long as the release index of at least one connected view is lower than $\texttt{i} - h + 1$, where h is the horizon of the recording. Then it increments the stall index of `r` to `i`. `void release(v, i)` increments the release index of `v` to `i`; it does nothing if `i` is lower than or equal to the current release index. The stall index is always lower than or equal to the minimum of the connected views' release indices. Initiation and termination issues are considered below.

[1] Unreasonable ones may queue inputs indefinitely; data-flow synchronous languages reject such programs using a clock calculus [9–11,22].

On Figure 1, index 0 has been released by the consumer, its value being (logically) removed from the view's buffer; it is not available for further computations. Because there is only one consumer, index 0 has also left the recording's buffer, making room for further value definitions and commits by the producer. The recording and view horizons are set to 4 and 2, respectively.

Sliding windows of recordings and views may be distinct, especially on distributed platforms. In such a case, it is safe for `update()` to implement the semantics of `receive()` and for `receive()` to be implemented as a no-op, anticipating the release of data and reducing the turn-around time of live elements.

The reader may wonder why back-pressure deserves dedicated primitives and is not implemented with `commit()` and `update()`, synchronizing with a shadow recording in the consumer and a shadow view in the producer. Modularity is the reason. The data-flow `commit()` and `update()` primitives require an explicit connection: using such primitives, a producer waiting for the release of indices by a consumer need to statically know to which consumer it is communicating with. This would violate the modularity of function composition, forcing the producer to be dedicated to a predefined collection of consumers. Our back-pressure design is a core component of ERBIUM's modular, split-phase communications.

Eventually, the representation of indices is another, subtle resource constraint. Applications will use unsigned 32-bit or 64-bit integers, depending on the target platform. We will assume 32-bit integers in the following. Overflow may occur, but infrequently enough so that performance is not impacted. The `commit()` primitive is responsible for preventing overflows, e.g., by detecting when the index crosses the 2^{31} All internal index variables (e.g., the current commit, update, release variables) can be translated backwards by the closest multiple of the horizon lower than or equal to the minimal release index. This translation is invisible to the sliding window indexing and preserves the non-negativeness of indices. Since the application itself refers to index variables, it is preferable to let those variable wrap-around safely without explicit control flow to detect overflows. This can be achieved by translating the index arguments transparently in the runtime implementation. The "translation bias" may start at 0 and be atomically updated along each threshold-crossing detected by `commit()`.

Creation and termination. A process is declared as a plain C function, introduced by the `process` keyword. It cannot be called as a function, and does not have a return value. The `run p(...)` spawns a new thread to run process `p`,[2] passing arbitrary arguments to initialize the new process instance, including recording arguments to communicate with other instances.

Initialization is a common source of complexity and deadlocks in concurrent applications. ERBIUM defines a standard, deterministic protocol, supporting modular composition, dynamic creation of processes and dynamic connection of views to recordings.

[2]ERBIUM may implement lightweight user-space threads and workstealing schedulers [33], but this is orthogonal to the scope of the paper.

`view T new_view(recording T r)` creates a fresh view descriptor, its update index initialized to 0, and connected to recording `r`.

`void register(view T v, int id)` adds `v` as the `id`-th *registered* view of its connected recording. This property controls which views are counted as part of the concurrent algorithm itself rather than independent observers: registered views are considered by the back-pressure mechanisms to avoid non-deterministic loss of data in case of late connection of the view. Non-registered views are also useful in asymmetric broadcasts where some consumers may safely miss parts of the data stream — such as transient or instrumentation processes. The registration property is a crude one: richer and more dynamic properties could be designed on the same basis.

`recording T new_recording(int m)` creates a fresh recording descriptor, its commit index initialized to 0. Argument `m` is the threshold number of *registered* views connected to the recording to unlock back-pressure mechanisms. More precisely, `stall(i)` may proceed if and only if all views of `id`s less than or equal to `m` have registered *and* the update index of *all registered views* is greater than or equal to `i`.

`void alloc(recording T r, int h)` and `void alloc(view T v, int h)` create a fresh sliding window of `h` elements of type `T` and attach it to `r` (resp. `v`). The horizon `h` is specific to a given recording or view. A process calling `alloc()` becomes the *owner* of the corresponding recording or view. This binds the sliding window buffers of recordings and views to a specific process instance. This binding is important when running Erbium on a distributed platform: the placement of data and process instances must be consistent. It is also useful for the recycling of process resources.

Recordings passed as arguments of processes must be initialized prior to spawning the process thread, or non-deterministic failures may happen when connecting views. Since the allocation of recording's buffers defines its owning process, the creation of the recording descriptor must be decoupled from the allocation of its buffer, hence the distinct `new_recording()` called prior to spawning the process instance.

Termination is at least as error-prone as initialization in concurrent applications, due to resource reclaiming (as illustrated by concurrent garbage collection). A process way free the recordings (resp. views) it owns through the `zombify()` primitive; it saves the last value of its commit (resp. update) index, then *waits until all connected views have released the corresponding index* (resp. have been notified of the termination), and finally frees the descriptor and buffer. For recordings, this *terminal* commit index is returned by the `update()` primitive, letting the consumer adjust to the situation where less elements than expected could be retrieved. The next call to `update()` returns 0 indicating proper termination. These two cases are the only ones where `update()` does not return the value of its second argument. An explicit or implicit `return` terminates a process.

Simple example. Figure 2 illustrates these concepts on a simple producer-consumer example. A single recording is connected to a single view. Data-flow synchronization, communication and back-pressure are

straightforward, with data element bursts in the $\{\texttt{tl}, \ldots, \texttt{hd}-1\}$ range. Each process sets its own recording and view horizon, and its own commit and update burst.

Notice that `commit()` follows the last definition of a value in the commit burst, `update()` precedes the first use of the update burst, `release()` follows the last use of the update burst, and `stall()` precedes the first definition of new indices in the commit burst.

Termination is detected in the consumer in two phases: first the return value of `update()` bounds the burst iteration to the precise number of retrieved elements, then control-flow breaks out of the loop at the next call.

The recording owned (allocated) by the producer is an initialization argument for the consumer; this recording's descriptor has been previously initialized prior to spawning the producer, and is used to connect the view to the recording in the consumer. Separate compilation of the producer and consumer is therefore possible.

This naive implementation is inefficient, especially on distributed memory: there is no overlap of communications with computations. A better version would add a prefetch distance to the call to `receive()`, looking a few data element ranges ahead. The distance would of course be platform-specific and tuned by the compiler or at runtime. The grain of synchronization can also be tuned, coarsening the burst size of the producer and/or consumer. Load-balancing can be achieved by adjusting the relative value of the burst sizes. Finally, on some high-latency systems like networks of workstations, it may simply not be beneficial to split the computation into distinct producer and consumer processes. In such a case, task fusion is the only solution and should be implemented by the compiler, statically scheduling the activations to avoid starvation and overflow.

This simple example can be trivially adapted into a broadcast with multiple consumers: the only changes are to run multiple consumers instead of one, and setting the minimal number of registered views accordingly when creating the recording descriptor.

4 Shared Memory Implementation

Let us discuss the specialization of Erbium on shared-memory platforms. Specialization for distributed-memory involves completely different algorithms, although these implementation differences are transparent to the Erbium-based application; this will be the purpose of a separate paper.

A recording or view is built of a data structure descriptor — holding the current indices and shared variables — and a separate sliding window buffer. The buffer is shared between the recording and its connected views. The size of the buffer is the sum of the recording horizon and the maximal view horizon; it is dynamically reallocated when connecting a view whose horizon exceeds the maximum size of all connected horizons.

The four synchronization primitives obey a non-blocking implementation, relying on as few hardware atomic operations as possible (and no software mutexes/locks). To avoid busy waiting on empty/full buffers, `update()` and `stall()` suspend the execution of the process. This involves *wait* and *wake/signal* operations, and is implemented in two different flavors: portable POSIX threads with condition variables and mutexes, and lower level Linux *futex*es [15]. All experiments use the futex version.

The code generator is implemented in an experimental branch of GCC 4.3. It expands the ERBIUM constructs to their shared-memory specializations *after* the main optimization passes. Task-level analyses and optimizations are not yet implemented. The most interesting step is to hide the concurrency constructs from the downstream optimizations. This can be achieved by systematically strip-mining bursts of computations enclosing the index-wise computations inside a protected inner loop. This approach was already proposed to support optimizations over streaming extensions of OpenMP [40]. It also removes modulo-indexing of sliding windows. This transformation was the most important step to enable automatic vectorization.

5 Experiments

Experiments target a 4-socket Intel hexa-core Xeon E7450 (Dunnington), with 24 cores at 2.4 *GHz*, and 4-socket AMD quad-core Opteron 8380 (Shanghai) with 16 cores at 2.5 *GHz*, both with 64 *GB* of memory. A distributed-memory implementation is also underway for the IBM Cell broadband engine. This port is not mature enough to conduct systematic experiments, but it was used to validate the portability of ERBIUM and confirm its very low memory footprint.

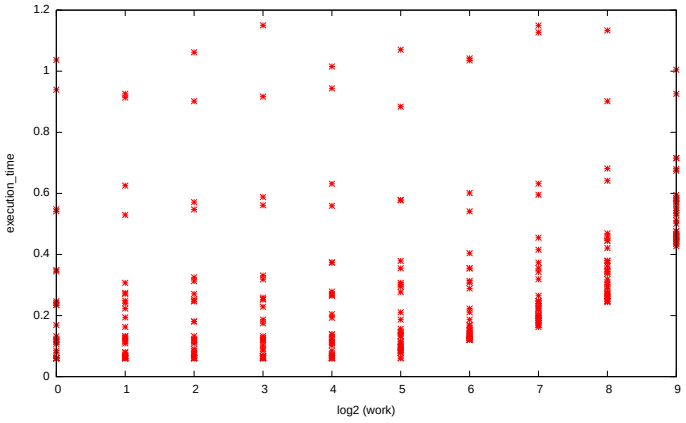

Figure 3: Tuning `exploration` on Xeon

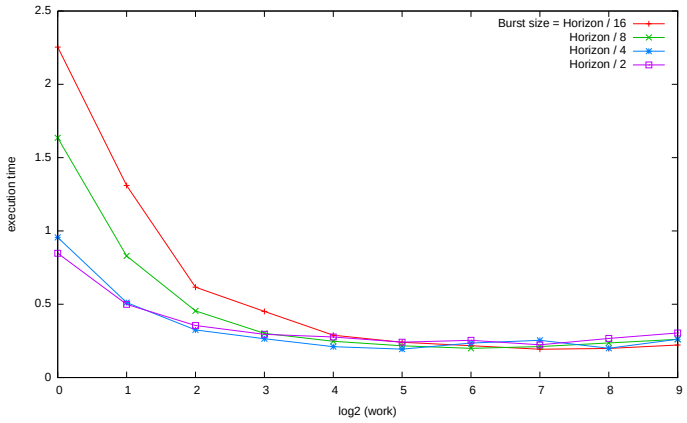

Figure 4: Synchronization cost

We studied a synthetic benchmark called `exploration`, with multiple producers broadcasting data to a larger number of consumers. Each index is associated with a fixed (parameterized) number of multiply-and-add operations. We report results on an extreme broadcast scenario, where n is the number of running process instances: 1 producer process (1 recording) and $n-1$ consumer processes (1 view per

process instance); this scenario stresses the scalability of the waiting roulette and of the estimation in the minimum release index in the back-pressure algorithm. In our experiments, n will take values in $\{4, 16, 24\}$ matching the number of hardware threads of our platforms.

Based on the fine-tuning achieved on the synthetic benchmarks, we wrote ERBIUM versions of three full applications: `fmradio` from the GNU radio package,[3] a `802.11a` production WiFi codec from Nokia,[4] and `jpeg`, a JPEG decoder rewritten in ERBIUM from a YAPI implementation of Philips Research [47]. They are all data streaming codes, representative of data crunching tasks running on single-node machines. It is complex enough to illustrate the expressiveness of ERBIUM, yet simpler than complete frameworks like h264 video that would require adaptive scheduling schemes not yet implemented in ERBIUM [5]. In addition, `802.11a` involves input-dependent mode changes and `jpeg` exhibits high variability in computation loads per macro-block.

5.1 Performance Tuning on a Synthetic Benchmark

Figure 3 report execution times for all configurations of the `exploration` benchmark. These configurations include varying horizon in a 256-to-4096 range, burst sizes in a horizon/16-to-horizon/2 range, and computational work per index in a 1-to-512 range. The scalability is excellent, even considering negligible amounts of work per index. But the lower the work amount, the larger the bursts should be to amortize synchronization, task waking and context-switch overhead.

Performance overhead is shown in Figure 4. This time, the total number of multiply-and-add operations is fixed: the total number of computed indices is proportional to the inverse of the amount of work per index. Interestingly, increasing the burst size alone is sufficient to bring the overhead to the minimum; this would hardly be possible with copy- and lock-based primitives.

In the extreme case of a single multiply-and-add per index, our implementation achieves 928491 index computations per second.[5] A previous lock-based implementation was almost two orders of magnitude slower.

5.2 Parallelization of Real Applications

We report experimental data about the parallelized version of our three benchmarks. `fmradio` and `802.11a` did not require algorithmic or radical design changes to achieve scalable performance; `jpeg` was written initially as a fine-grain KPN and only required systematic coarsening of the comminications.

On `fmradio`, exploiting task and pipeline parallelism is straightforward but shows limited scalability — 6 concurrent processes. Exploiting data parallelism is not trivial and involves an interesting transformation:

[3] `http://gnuradio.org/trac`

[4] From the ACOTES FP6 European project.

[5] One index computation corresponds to a cycle of the 4 synchronization primitives and the communication of one value.

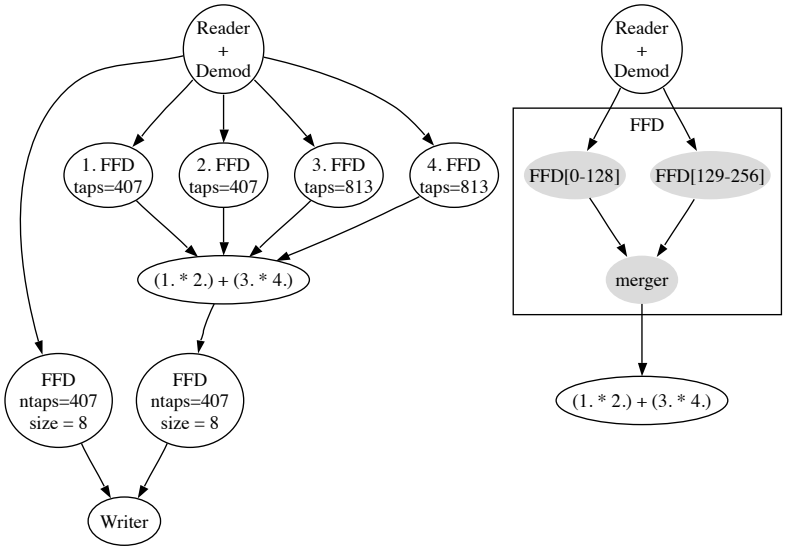

Figure 5: Informal data flow of `fmradio`

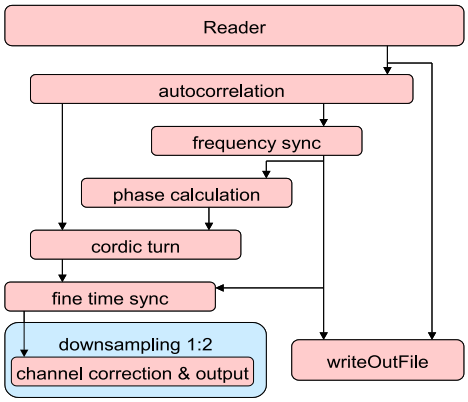

Figure 6: Informal data flow of `802.11a`

Platform (cores)	Seq. `-O3`	Par. `-O2`	Par. `-O3`	Par. `-O3` vs. Par. `-O2`
Xeon (24)	1.14	10.1	12.6	1.25
Opteron (16)	1.52	9.51	14.6	1.54

Figure 7: Speedups results for `fmradio`

Platform (cores)	Task-Level Only	No Decoupling	Combined Parallelism
Xeon (24)	1.85	1.84	6.67
Opteron (16)	2.73	2.81	7.45

Figure 8: Speedups results for `802.11a`

the original code uses a circular window using modulo arithmetic and holding the results of previous filtering iterations; it can be replaced by a recording, removing spurious memory-based dependences.

Figure 5 illustrates the concurrency exposed in `fmradio`. On the left, 4 FFD processes process the signal, implementing different sub-samplings and linear transforms (amplitude- and frequency-domain). They account for most of the computation load. Two of them operate at twice the sampling rate of the two others, involving twice the number of "taps" and twice as many computations. This suggests to balance the load by creating twice as many instances for the heavier ones. The right side of the figure details the data-parallelization of an FFD process, sharing the work into two instances. Figure 7 summarizes the speedups achieved with GCC 4.3 and different optimization options. The baseline is the sequential (original) version compiled with `-O2` (no vectorization, less optimizations); it runs in 13.65 s on Xeon. These results confirm the scalability of Erbium on a real application; they also confirm its compiler-friendliness, with the automatic vectorizer of GCC capable of aggressive loop restructuring in presence of concurrency primitives, recording and view accesses.

Figure 6 illustrates the concurrency exposed in `802.11a`. The data-flow graph is more unbalanced than `fmradio`; it is also very far from the fork-join graphs of StreamIt [49]. We do not take I/O tasks into account. All the remaining tasks take a significant fraction of the total execution time. Among those, `frequency_sync` and `fine_time_sync` *cannot* be data-parallelized easily: they need to be further decoupled into a sequential fast-forward loop and a data-parallel kernel [37]. Figure 7 displays speedup results. It demonstrates the performance advantage of combining task-level and data parallelism (*Combined*

Parallelism), compared with the partially data-parallel one (*No Decoupling*, keeping the same degree of parallelism for the remaining data-parallel tasks) and task-level parallelism only. Partial data-parallelization even degrades performance on Xeon due to work-sharing overheads.

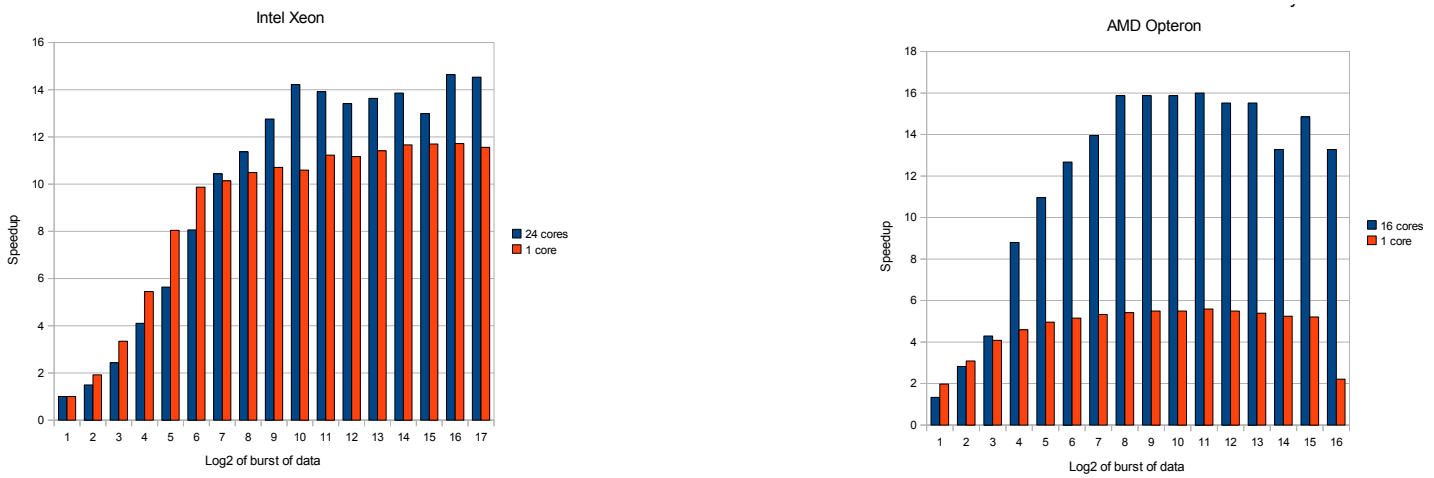

Figure 9: Performance of `jpeg` function of synchronization grain size

On `jpeg`, the systematic macro-block-level decomposition of the application exposes 26 fine grain tasks. Most of these can be further data-parallelized, but we choose to limit ourselves to a pipelined and task-parallel version: the objective of this experiment is to confirm our results on the synthetic benchmark and demonstrate the benefits of the low-cost abstractions of ERBIUM in the real world. Figure 9 show the impact on the synchronization grain. Larger bursts are required to reach the performance plateau when running on all cores. Also, increasing the burst size eventually yields performance degradation when running out of one or both cache levels. Speedup reaches 2.42× on Xeon and 1.95× on Opteron for the optimal grain, exploiting task-level parallelism only. These numbers are low but they confirm that ERBIUM succeeds in exploiting fine-grain task parallelism on real applications, despite the high rate of synchronizations. This is encouraging about the scalability of bandwidth-bound applications on future manycore architectures, when data-parallelism alone does not scale.

Streaming codes are often bandwidth-bound. The Xeon's front-side bus is clearly penalized on such codes compared to the Opteron's Hypertransport busses: data-parallelism is limitted by off-chip memory bandwidth. Indeed, compared to a pure task-level parallel (pipelined) implementation, data parallelism may offer better scalability; and compared to a pure data-parallel implementation, pipelining reduces memory bandwidth contention and amortizes load imbalance (absence of synchronization barrier). The tradeoff between task and data-level parallelism depends on the target architecture. Notice that pipelining also increases expressiveness, extracting parallelism from dependent iterations in a loop, as in `802.11a`. Overall, ERBIUM leverages much more flexible, scalable and efficient forms parallelism than restricted models.

5.3 Comparison With Lightweight Scheduling of Short Tasks

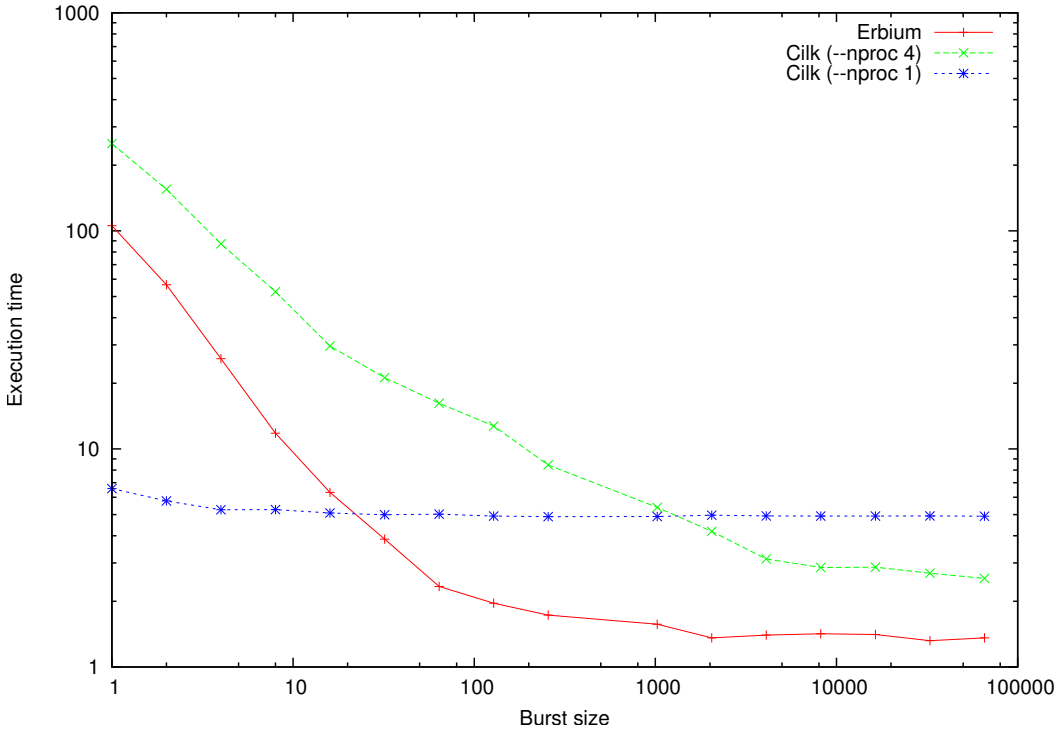

Figure 10: Long-lived processes vs. short-lived atomic tasks

Erbium favors long-running tasks with iterated lightweight synchronizations. This differs from a more common approach where concurrency is expressed at the level of atomic, short running tasks. `exploration` synthetic benchmark with a Cilk implementation spawning short-lived user-level tasks [33]. The target machine is a 4-core Intel Core 2 Duo desktop, and Cilk is run with the `--nproc 4` option to generate parallel code, and with the `--nproc 1` option to specialize the code for sequential execution. The baseline sequential execution takes almost 7 s for the finest synchronization grain, and 5 s for larger ones. The parallel Cilk version with the finest synchronization takes 221.4 s and the corresponding Erbium version takes 107.7 s. The performance gap widens significantly for intermediate size bursts, and reaches almost 5× when the Erbium version reaches its performance plateau. But the most important figure in practice is that the Erbium version breaks even for grain size 80× smaller than Cilk. It demonstrates the need for data-flow interactions among long-lived processes as an essential abstraction for scalable concurrency. Short-lived atomic tasks may be better supported with dedicated hardware [29]. Nevertheless, lightweight threading techniques are still very useful for load balancing and to increase the reactivity of passive synchronizations (blocking `update()`/`stall()` on empty/full buffer).

6 Related Work

Concurrency models have been designed for maximal expressiveness and generality [24,34], with language counterparts such as Occam [12]. Asynchronous versions have been proposed to simplify the implementation on distributed platforms and increase performance [18,25,35], with language counterparts such as JoCaml [17]. Our goal is different: parallelism is only a specialization and optimization of ERBIUM. We need concurrency if it is useful for exploiting parallelism on some target platform. The data-flow concurrency expressed in our model is sufficient to expose scalable parallelism in a wide spectrum of applications; it also offers strong determinism and liveness guarantees that evade more expressive models.

Our work is strongly influenced by data-flow and streaming languages, including Id and I-Structures [3], SISAL [27], Lustre [22], Lucid Synchrone [9], Jade [44] and StreamIt [48,49]. These languages share a common interest in determinism (time-independence) and abstraction. They also involve advanced compilation techniques, including static analysis to map declarative semantics to effective parallelism, task-level optimizations and static scheduling. The intermediate representation of ERBIUM is an ideal target for a front-end compiler for such abstract languages and for implementing platform-specific optimizations. But unlike the previous proposals, it can also be used for highly efficient infrastructure developments.

Extensions of OpenMP have been proposed to improve the support for pipeline parallelism and streaming applications [7,40]. These are promising tradeoffs between declarative abstractions and explicit, target-specific parallelization, aiming for an incremental introduction of richer forms of concurrency and expression of locality in programming languages. They do suffer, however from expressiveness limitations: the `fmradio` application has been initially parallelized with such approaches, but data parallelism could not easily be expressed. Pop et al. report speedup saturating around 3× on 4-core to 16-core x86-64 platforms [40].

High-level and domain specific libraries have been proposed to shield the programmers from concurrency subtleties, raising the level of abstraction [2,19,45]. These approaches are complementary to ERBIUM as higher level abstractions. But they also suffer from expressiveness limitations (algorithmic skeletons) and they induce abstraction penalty not easily managed by the C++ compiler front-ends (including code size overhead unacceptable in embedded and memory-constrained accelerators). Unlike the former approaches, some of these libraries choose to trade safety for the ability to express non-deterministic concurrency [2,45]; they fill a gap in the domain of concurrent data structures that ERBIUM is unable to address as efficiently today. This motivates further research in combining data-flow and transactional concurrency in a single, consistent programming model.

Our approach is complementary to the large body of work in lightweight runtimes. Cilk does not natively support data-flow concurrency [33] but we share its emphasis on compile-time specialization. We wish to integrate its work stealing and load-balancing capabilities in the future, as motivated by Azevedo et al. for streaming applications [5]. In addition, ERBIUM binds processes and data together, facilitating

process migration and fault tolerance. Regarding the distributed-memory implementation of ERBIUM, we will leverage the results of proposals like StarSs (data-flow oriented language, [38]) and StarPU (process network, runtime approach, [4]).

Further performance improvements can be achieved with hardware support, starting from the pioneering data-flow architecture designs [14,50], recently revived for coarser-grain, task-level concurrency [1,29,46]. Hardware support reduces the turn-around time for context-switch and thread awaking.

Finally, our work is totally independent from verification-oriented enforcement of deterministic execution in the scheduler [36,41]. These approaches do not define a deterministic semantics for the application, independently of the architecture.

7 Conclusion

We introduced ERBIUM and its three main ingredients: an intermediate representation for compilers and efficiency programmers, a data structure for scalable and deterministic concurrency, and a lightweight runtime. The intermediate representation is being implemented in GCC 4.3 and allows classical optimizations and parallelizing transformations to operate transparently. It relies on 4 concurrency primitives implemented with platform-specific, non-blocking algorithms. The data structure called *Event Recording* (Er) is the basis for a scalable and deterministic concurrency model with predictable resource management. Our current implementation has a very low footprint and demonstrates high scalability and performance. Unlike usual runtime approaches to low-level parallel programming, the intermediate representation *is* the portability layer. We are working on porting ERBIUM to distributed memory machines (Cell BE and clusters) and on front-ends for high-level languages.

References

[1] G. Al-Kadi and A. S. Terechko. A hardware task scheduler for embedded video processing. In *Proc. of the 4th Intl. Conf. on High Performance and Embedded Architectures and Compilers (HiPEAC'09)*, Paphos, Cyprus, Jan. 2009.

[2] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. M. Amato, and L. Rauchwerger. Stapl: An adaptive, generic parallel C++ library. In *Languages and Compilers for Parallel Computing (LCPC'01)*, pages 193–208, 2001.

[3] Arvind, R. S. Nikhil, and K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. on Programming Languages and Systems*, 11(4):598–632, 1989.

[4] C. Augonnet, S. Thibault, R. Namyst, and M. Nijhuis. Exploiting the Cell/BE architecture with the StarPU unified runtime system. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS'09)*, pages 329–339, 2009.

[5] A. Azevedo, C. Meenderinck, B. H. H. Juurlink, A. Terechko, J. Hoogerbrugge, M. Alvarez, and A. Ramírez. Parallel H.264 decoding on an embedded multicore processor. In *Proc. of the 4th Intl. Conf. on High Performance and Embedded Architectures and Compilers (HiPEAC'09)*, Paphos, Cyprus, Jan. 2009.

[6] G. Bilsen, M. Engels, L. R., and J. A. Peperstraete. Cyclo-static data flow. In *Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP'95)*, pages 3255–3258, Detroit, Michigan, May 1995.

[7] P. M. Carpenter, D. Ródenas, X. Martorell, A. Ramírez, and E. Ayguadé. A streaming machine description and programming model. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS'07)*, pages 107–116, Samos, Greece, July 2007.

[8] P. Caspi, J.-L. Colaço, L. Gérard, M. Pouzet, and P. Raymond. Synchronous objects with scheduling policies: introducing safe shared memory in lustre. In *LCTES*, pages 11–20, 2009.

[9] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *ACM Intl. Conf. on Functional programming (ICFP'96)*, pages 226–238, 1996.

[10] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. N-Sychronous Kahn networks. In *ACM Symp. on Principles of Programming Languages (POPL'06)*, pages 180–193, Charleston, South Carolina, Jan. 2006.

[11] A. Cohen, L. Mandel, F. Plateau, and M. Pouzet. Abstraction of clocks in synchronous data-flow systems. In *6th Asian Symp. on Programming Languages and Systems (APLAS 08)*, Bangalore, India, Dec. 2008.

[12] I. Corp. *Occam Programming Manual*. Prentice Hall, 1984.

[13] D. E. Culler and Arvind. Resource requirements of dataflow programs. In *ISCA*, pages 141–150, 1988.

[14] J. B. Dennis and G. R. Gao. An efficient pipelined dataflow processor architecture. In *Supercomputing (SC'88)*, pages 368–373, 1988.

[15] U. Drepper. Futexes are tricky. http://people.redhat.com/drepper/futex.pdf, Aug. 2009. Latest revision.

[16] K. Fatahlian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Supercomputing 2006*, Tampa, Florida, Nov. 2006.

[17] F. L. Fessant and L. Maranget. Compiling join-patterns. *Electr. Notes Theor. Comput. Sci.*, 16(3), 1998.

[18] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, Jan. 1996. ACM.

[19] A. Ghuloum, T. Smith, G. Wu, X. Zhou, J. Fang, P. Guo, B. So, M. Rajagopalan, Y. Chen, and B. Chen. Future-proof data parallel algorithms and software on Intel multi-core architecture. *Intel Technology Journal*, Nov. 2007.

[20] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, San Jose, California, 2006.

[21] R. Gupta. Exploiting parallelism on a fine-grain MIMD architecture based upon channel queues. *Intl. J. of Parallel Programming*, 21(3):169–192, 1992.

[22] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, Sept. 1991.

[23] R. H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. on Programming Languages and Systems*, 7(4):501–538, 1985.

[24] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[25] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, pages 133–147. Springer-Verlag, 1991.

[26] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug. 1974. North Holland, Amsterdam.

[27] C. Kim, J.-L. Gaudiot, and W. Proskurowski. Parallel computing with the sisal applicative language: Programmability and performance issues. *Software, Practice and Experience*, 26(9):1025–1051, 1996.

[28] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *ACM Conf. on Programming Language Design and Implementation (PLDI'08)*, pages 114–124, June 2008.

[29] C. Kyriacou, P. Evripidou, and P. Trancoso. Data-driven multithreading using conventional microprocessors. *IEEE Trans. on Parallel Distributed Systems*, 17(10):1176–1188, 2006.

[30] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, 36(1):24–25, 1987.

[31] E. A. Lee and A. L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.

[32] R. Lublinerman, C. Szegedy, and S. Tripakis. Modular code generation from synchronous block diagrams: modularity vs. code size. In *ACM Symp. on Principles of programming languages (POPL'09)*, pages 78–89. ACM, 2009.

[33] K. H. R. M. Frigo, C. E. Leiserson. The implementation of the Cilk-5 multithreaded language. In *ACM Symp. on Programming Language Design and Implementation (PLDI'98)*, pages 212–223, Montreal, Quebec, June 1998.

[34] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i and ii. *Inf. Comput.*, 100(1):1–40 and 41–77, 1992.

[35] U. Nestmann and B. C. Pierce. Decoding choice encodings. *Inf. Comput.*, 163(1):1–59, 2000.

[36] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *The International Conference on Architectural Support for Programming Languages and Operating Systems*, Washington, DC, Mar 2009.

[37] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *IEEE Intl. Symp. on Microarchitecture (MICRO'05)*, pages 105–118, 2005.

[38] J. M. Pérez, P. Bellens, R. M. Badia, and J. Labarta. CellSs: Making it easier to program the cell broadband engine processor. *IBM Journal of Research and Development*, 51(5):593–604, 2007.

[39] K. Pingali and Arvind. Efficient demand-driven evaluation - Part 1. *ACM Trans. on Programming Languages and Systems*, 7(2):311–333, 1985.

[40] A. Pop, S. Pop, and J. Sjödin. Automatic streamization in GCC. In *GCC Developer's Summit*, Montreal, Quebec, June 2009.

[41] M. Prvulovic and J. Torrellas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *ISCA*, pages 110–121, 2003.

[42] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *ACM Intl. Symp. on Code Generation and Optimization (CGO'08)*, pages 114–123, Apr. 2008.

[43] M. Ren, J. Y. Park, M. Houston, A. Aiken, and W. J. Dally. A tuning framework for software-managed memory hierarchies. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'08)*, pages 280–291. ACM Press, 2008.

[44] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Trans. on Programming Languages and Systems*, 20(3):483–545, 1998.

[45] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in TBB. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS'08)*, pages 1–8, Miami, Florida, Apr. 2008.

[46] M. Själander, A. Terechko, and M. Duranton. A look-ahead task management unit for embedded multi-core architectures. In *Proc. of the 2008 11th EUROMICRO Conf. on Digital System Design Architectures*, Parma, Italy, Sept. 2008.

[47] S. Stuijk. Concurrency in computational networks. Master's thesis, Technische Universiteit Eindhoven (TU/e), Oct. 2002. # 446407.

[48] W. Thies. *Language and Compiler Support for Stream Programs*. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, Feb 2009.

[49] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Intl. Conf. on Compiler Construction*, Grenoble, France, Apr. 2002.

[50] I. Watson and J. R. Gurd. A practical data flow computer. *IEEE Computer*, 15(2):51–57, 1982.