Erbium: A Deterministic, Concurrent Intermediate Representation to Map Data-Flow Tasks to Scalable, Persistent Streaming Processes

Cupertino Miranda, Antoniu Pop, Philippe Dumont, Albert Cohen, Marc Duranton

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Erbium: A Deterministic, Concurrent Intermediate Representation to Map Data-Flow Tasks to Scalable, Persistent Streaming Processes

Cupertino Miranda
INRIA Saclay and LRI, Paris-Sud
11 University
Cupertino.Miranda@inria.fr

Antoniu Pop
CRI, MINES ParisTech
Antoniu.Pop@cri.ensmp.fr

Philippe Dumont
INRIA Saclay and LRI, Paris-Sud
11 University
Philippe.Dumont@inria.fr

Albert Cohen
INRIA Saclay and LRI, Paris-Sud
11 University
Albert.Cohen@inria.fr

Marc Duranton
CEA, LIST, Laboratoire Calcul
Embarqué
Marc.Duranton@cea.fr

ABSTRACT

Tuning applications for multicore systems involve subtle concurrency concepts and target-dependent optimizations. This paper advocates for a streaming execution model, called ER-BIUM, where persistent processes communicate and synchronize through a multi-consumer multi-producer sliding window. Considering media and signal processing applications, we demonstrate the scalability and efficiency advantages of streaming compared to data-driven scheduling. To exploit these benefits in compilers for parallel languages, we propose an intermediate representation enabling the compilation of data-flow tasks into streaming processes. This intermediate representation also facilitates the application of classical compiler optimizations to concurrent programs.

Categories and Subject Descriptors

D.3.4 [**Programming Langugages**]: Compilers

General Terms

Performance, Languages, Algorithms

1. INTRODUCTION

High-level parallel languages capture (in)dependence and locality properties without reference to any particular hardware. Compilers and runtime systems are responsible for lowering these abstractions to well-orchestrated threads and memory management, and for implementing target-specific optimizations. Parallel stream and data-flow programming makes the task-level data-flow explicit. It exposes pipeline,

data and task parallelism with a functional determinism guarantee: a major asset in the quest for productivity. An intermediate compiler representation is required to implement the specialization steps converting portable data-flow programs into efficient parallel implementations. Among these necessary steps, static scheduling plays a major role: it coarsens the grain of dynamic scheduling and synchronization, adapting it to the target platform (bandwidth, latency, local memory). Coarser grain tasks require a dedicated synchronization and communication mechanism: they continuously exchange data through FIFO streams. Without such data streams, static scheduling may not be possible because of dependence cycles. Streaming communications help reduce the severity of the memory wall: decoupled producer-consumer pipelines naturally hide memory latency and favor local, on-chip communications, bypassing global memory, a big advantage on MPSoC architectures.

This paper advocates for persistent, long-running streaming processes and communications. Our approach is complementary to lightweight scheduling: streaming is the preferred execution model for the finest grain of thread-level parallelism, while data-driven scheduling deals with load-balancing and dynamic task invocation at coarser grain. The core of the approach, called ERBIUM, is a data structure for scalable and deterministic concurrency, an intermediate representation for compilers, a low-level language for efficiency programmers, and a fast runtime implementation.

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 in Section 7.

2. DESIGN

ERBIUM defines an intermediate representation for compilers, also usable as a low-level language for efficiency programmers. Its features a unique combination of productivity and performance properties.

Determinism. ERBIUM's semantics derives from *Kahn Process Networks* (KPNs) [22]. KPNs are canonical concurrent extensions of (sequential) recursive functions preserving determinism (a.k.a. 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 [22,26]. The 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.

Expressiveness. Parallelism is often implicit in high-level data-flow languages [3,23]. As an intermediate representation, ERBIUM is explicitly parallel. ERBIUM supports dynamic creation, termination of concurrent processes.

ERBIUM favors persistent, long-running processes communicating through point-to-point data streams. Traditional streaming communications involve `push()` and `pop()` primitives over FIFO channels. ERBIUM's data structure for communication is much richer: it provides random-access *peek* (read), *poke* (write) and communications decoupled from the actual synchronization. This abstract data structure is called an *event record.*[1] It unifies *streams* and *futures* [19], and generalizes them to support multiple producers and multiple consumers. We rest on the programming language semantics and on the compiler in charge of lowering high-level abstractions to ERBIUM.

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 concurrent processes dynamically. Since ERBIUM provides explicit means to manage resources (e.g., communication buffers), it puts a specific challenge on the ability to compose processes in a modular fashion. To this end, we introduce a low-level mechanism for modular back-pressure supporting arbitrary broadcast and work-sharing scenarios.

Static adaptation. As an intermediate language, ERBIUM supports specialization, static analysis and optimization. Binary code does not offer the required level of static adaptation; on the contrary ERBIUM *defines the intermediate representation as the portability layer.*

The ERBIUM runtime may transparently be specialized for different memory models. In the following, we assume a shared, global address space whose caches are kept coherent in hardware. Specialization for distributed-memory involves completely different algorithms; this will be the purpose of a separate paper. In addition, ERBIUM may transparently exploit any hardware acceleration for faster context switch [1,24,37], synchronization and communication [16,31].

The compiler selects the most relevant hardware operations and inlines ERBIUM's split-phase communication primitives. It may also adapt the grain of concurrency through task-level and loop transformations.

---

[1]Abbreviated as Er... symbol of the Erbium element.

Lightweight, efficient implementation. ERBIUM aims to be closest to the hardware while preserving portability and determinism. Any overhead intrinsic to its design and any implementation overhead hits scalability and performance; such overheads cannot be recovered by a programmer who operates at this or higher levels of abstraction.

Thanks to its data-flow semantics, it is possible to implement the primitives of the ERBIUM runtime only relying on non-blocking synchronizations. Leveraging ERBIUM's native support for multiple producers and multiple consumers, broadcast and work-sharing patterns can be implemented very efficiently, avoiding unnecessary copy in (collective) scatter and gather operations.

The split-phase communication approach hides latency without thread scheduling or switching overhead: it relies entirely on existing hardware such as prefetching or DMA.

3. SEMANTICS

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

Figure 1 shows the ERBIUM primitives and event record structures on a producer-consumer template. This example illustrates data-flow synchronization and communication, resource management, process creation and termination.

```
int main() {
  record int re = new_record(1, 1);
  run producer(re); run consumer(re);
}
process producer (record int re) {
  int tl=0, hd, i;
  view int vi = new_write_view(re);
  register (vi);
  alloc(vi, P_HORIZ);
  while (1) {
    hd = tl + P_BURST;
    if (hd>=N) break;
    stall(vi, hd);
    for (i=tl; i<hd; i++)
      vi[[i]] = foo(i);
    commit(vi, hd);
    tl = hd;
  }
}
```

```
process consumer (record int re) {
  int tl=0, hd, i;
  int sum=0;
  view int vi = new_read_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

Data streaming.

Processes communicate through a concurrent data structure called *event record*, or *record* for short. Effective communication and synchronization take place through *read* and *write views* connected to a record. A read or write view is an unbounded stream, randomly addressable through non-negative *indices*. Each read (resp. write) view is associated with a private, monotonically increasing *update* (resp. *commit*) *index*. Read view elements are read-only. Read view elements at indices less than or equal to the update index are identical to the corresponding elements of the connected record. Write view elements at indices less than or equal to the commit index are read-only.

`record T r` (resp. `view T v`) declares a record `r` (resp. view `v`) of data elements of type `T`. The `[[i]]` syntax is used to subscript a view at index `i`.

On Figure 1, the producer process committed indices 0, 1 and 2 to the write view, 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 2, 1 and below are read-only. On the consumer side, only indices 0 and 1
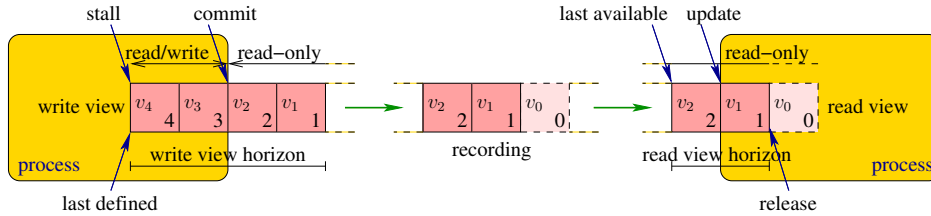
Figure 1: Producer-consumer flow

have been updated into the view; index 2 is available but the consumer did not yet decide to observe it in the view.

The `commit()` and `update()` primitives implement data-flow *pressure*, enforcing causality among processes.

- `void commit(view T v, int i)` increments the commit index of `v` 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 minimum commit index of connected write views is greater than or equal to `i`. It returns the value of `i`, except on deallocation of the connected record, to be explained in the termination section.

`update()` controls which 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 minimum commit index of its connected write views. 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.

ERBIUM follows a split-phase design, decoupling communication from synchronization: `update()` does not deal with communication, which is asynchronous and initiated with the `receive()` primitive.

- `void receive(view T v, int i)` updates `v` with the data of its connected record, starting from the current update index up to `i`. It cannot complete until the corresponding commit index reaches `i`; but the communication may start earlier, retrieving sub-ranges of indices, this is left to the implementation and not observable at the level of the intermediate representation.

- `void receive_range(view T v, int tl, int hd)` is a variant to retrieve indices in the $\{tl+1, \ldots, hd\}$ range. This primitive allows to subsample data streams and to distribute data among workers.

When an asynchronous call to `receive(v, i)` is pending, a follow-up `update(v, j)` must wait until all elements of indices lower than $\min(i, j)$ have been retrieved. On shared-memory platforms, `receive()` may be implemented as prefetch or no-op. Although the call is asynchronous, it may be easier/preferable on some embedded targets to block until the commit index satisfies the condition, before triggering the communication.

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 records need a bounded memory space. The bound may be managed statically or dynamically, and corresponds to the maximal number of live elements. The live elements of a read or write view are stored in a *sliding window*, and its size is called the view's *horizon*. It is not allowed to access elements outside the horizon. As with any blocking write semantics, KPN with bounded buffers may suffer from resource deadlocks when the view horizon is insufficient. In general, buffer size inference or adaptation is the responsibility of the compiler [8,25] and/or of the runtime library [10].

The `release()` and `stall()` primitives implement *back-pressure*. An element is considered live in a read view as long as `release()` has not been called on a higher index. An element is considered live in a write view as long as at least one connected read view has not yet released its index, calling `release()` on a higher index. Each write view (resp. read view) is associated with a private, monotonically increasing *stall* (resp. *release*) index, marking the tail of live elements in the write view. `void stall(view T v, int i)` waits as long as the release index of one or more connected read views is lower than or equal to $i - h$, where h is the horizon of the write view `v`. Then it increments the stall index of `v` to `i`. The storage locations for elements of indices lower than $i - h$ can now be recycled. `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 read views' release indices.

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 write view's buffer, making room for further value definitions and commits by the producer. The write and read view horizons are set to 4 and 2, respectively.

The representation of indices is a subtle resource constraint. 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 by detecting when the index crosses the 2^{31} limit. 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. The application does not have to be aware of this translation. The index variables

it uses can wrap-around safely, as the runtime primitives can transparently translating index arguments.

Creation.

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`, passing arbitrary arguments to initialize the new process instance, including record 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 process creation and connection of views.

`record T new_record(int wreg, int rreg)` creates a fresh descriptor for a record. The role of arguments `wreg` and `rreg` will be described below.

`view T new_read_view(record T r)` creates a read-only view descriptor connected to record `r` and initializes its update and release indices to 0.

Similarly, `view T new_write_view(record T r)` creates a read/write view descriptor connected to `r` and initializes its commit and stall indices to 0.

`int register(view T v, int id)` sets `v` as a registered view of its connected record, assigning it the `id` label. `id`s must be consecutive positive integers. Registered views are handled specially: `update(i)` (resp. `stall(i)`) may proceed if all views of `id`s less than or equal to `rreg` (resp. `wreg`) have registered *and* the update (resp. stall) indices of *all connected* read (resp. write) views are greater than or equal to `i`. Registration avoids non-deterministic loss of data in case of late connection of a view. It also enables deterministic hand-over of communications from one view to another with the same `id`.

Conversely, non-registered views are useful in asymmetric broadcasts where some consumers may safely miss parts of the data stream — such as observation or instrumentation processes. The call (`register(v, -1)`) unregisters `v`.

`void alloc(view T v, int h)` create a fresh sliding window of `h` elements of type `T` and attach it to `v`. The horizon `h` is specific to a given view.

Termination.

Termination is at least as error-prone as initialization in concurrent applications. A process may deallocate a view it owns through the `free_view()` primitive. The terminal commit (resp. release) index of a deallocated view is stored, and used in the computation of the minimal commit (resp. release) index. The view is disconnected from its recording and its resources are freed when its terminal commit (resp. release) index has been passed by the update (resp. stall) indices of views it is connected to. In case the deallocated view was registered, the `free_view()` primitive can be used with a termination argument that forbids hand-over of communication, i.e., the registration of a view with the same `id`. In the case of a write view, when the minimal commit index is held by a deallocated view marked with this termination argument, `update()` does not wait and may return a different value from its index argument. A following `update()` call returns 0, indicating proper termination of operations through a given record. Termination is the only case where `update()` does not return the value of its second

argument. Termination does not impact `stall()` directly: back-pressure is not meant to carry semantical information.

When `free_view()` is called for the last view connected to a given record, the latter is deallocated right after deallocation all the remaining data from former connected views. An explicit or implicit `return` terminates a process.

Modular back-pressure.

The registration mechanism above is essential to achieving modular composition. But the reader may wonder more specifically why back-pressure is not implemented with `commit()` and `update()` on "shadow" views. Modularity is the reason. `stall()` and `release()` use an a previously connected view, whereas `commit()` and `update()` would require a new connection: a producer waiting for the release of indices by a consumer would need to statically know to which consumer it is communicating with. This would violate the modularity of function composition, dedicating the producer to a predefined collection of consumers.

Simple example.

Figure 2 illustrates these concepts on a producer-consumer example. A single record is connected to a pair of read and write views. Data-flow synchronization, communication and back-pressure are straightforward, with index and data bursts in the $\{tl + 1, \ldots, hd\}$ range. Each process sets its own 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 record descriptor, defined and initialized in the *main* function is passed as argument to both processes. Both views connect to it in their respective process. This makes separate compilation of the producer and consumer possible.

This naive implementation is inefficient. 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 it may simply not be beneficial to split the computation into distinct producer and consumer processes: task fusion may be the only solution and should be implemented by the compiler, statically scheduling the activations while avoiding 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, setting the appropriate view `id`s, and setting the minimal number of registered views accordingly when creating the record descriptor. Likewise it is also easy to distribute work across writers or readers, using a stride to separate accesses into disjoint index ranges.

4. SHARED MEMORY TARGETS

On a distributed memory target, each view holds a private sliding window buffer, and communications involve explicit copies across these private buffers. In such a case, notice that calls to `release()` can be made redundant: `update()` can implement an early release, reducing the turn-around time of live elements. `release()` becomes essential in a shared memory implementation, where it is possible to allocate a single buffer shared between the connected write and read views. The size of such a buffer is the sum of the maximal write view horizon and the maximal read view horizon; it is dynamically reallocated when connecting a view whose horizon exceeds the maximum size of all connected horizons.

The four synchronization primitives can be implemented very efficiently on standard cache-coherent hardware. Considering the Total Store Ordering (TSO) memory model of x86 and SPARC ISAs, our implementation does not require any memory fence or atomic instruction (such as compare-and-swap, test-and-set, fetch-and-add). Our algorithm also minimizes cache line invalidation and on-chip contention. We revisited a lock-free implementation of a sliding-window to support multiple producers and multiple consumers [15]. This extension is made possible by delegating the task of index-range negotiation to the compiler. The synchronization primitives are left with a simpler concurrency problem. This algorithm and its implementation will be presented in another paper.

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. The synchronization primitives are preserved as "builtin" functions. We leveraged the expressive attribute and builtin mechanisms in GCC to expose their side-effects to the data-flow analyses of the compiler. The goal is to preserve the scalar and loop optimizations in the middle-end of the compiler from inline assembly, external function calls and pointers, hence to retain all their aggressiveness. But the synchronization primitives are still hampering some optimizations, such as automatic vectorization: as a result, we systematically strip-mine 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 [35]. It also removes modulo-indexing of sliding windows. We will see the impact of this design in the experimental evaluation. A library-based alternative would inhibit many compiler optimizations, as it is currently the case with early OpenMP expansion in GCC [35].

5. EXPERIMENTS

Experiments target a 4-socket Intel hexa-core Xeon E7450 (Dunnington), with 24 cores at 2.4 *GHz*, a 4-socket AMD quad-core Opteron 8380 (Shanghai) with 16 cores at 2.5 *GHz*, both with 64 *GB* of memory, and an Intel quad-core Core 2 Q9550 at 2.83GHz. These targets are respectively called Xeon, Opteron and Core 2 in the following.

We studied a synthetic benchmark called `exploration`, with multiple producers broadcasting data to multiple consumers. Each process implements a simple loop enclosing a pair of synchronization primitives updating, stalling, committing and releasing a burst of k indices at every iteration. The workload for each index amounts to a single integer load (consumer side) or store (producer side) only.
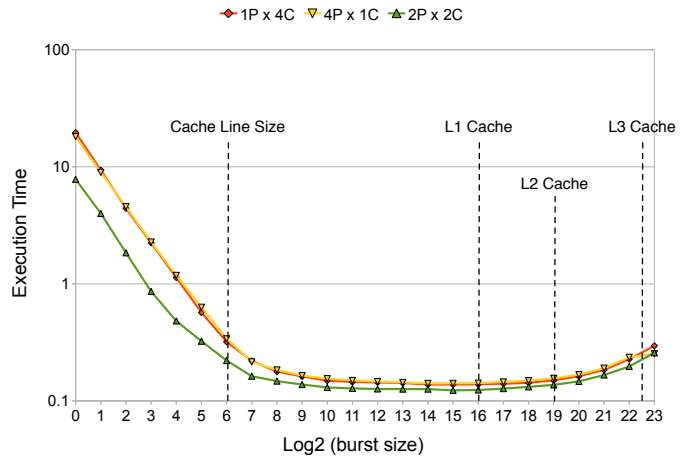

Figure 3: Burst size impact on Opteron

We then wrote ERBIUM versions of one classical signal-processing kernel and three full applications: `fft` from the StreamIt benchmarks [40], `fmradio` from the GNU radio package and also available in the StreamIt benchmarks, a `802.11a` code from Nokia [12], and `jpeg`, a JPEG decoder rewritten in ERBIUM from a YAPI implementation of Philips Research [39]. They are representative of data crunching tasks running on both general-purpose and embedded platforms. These applications are complex enough to illustrate the expressiveness of ERBIUM, yet simpler than complete frameworks like H.264 video that would require adaptive scheduling schemes not yet implemented in ERBIUM [5]. In addition, `802.11a` involves input-dependent mode changes that do not fit the expressiveness constraints of StreamIt, `jpeg` has variable computation load per macro-block and both `jpeg` and `fft` feature very fine grain tasks.

5.1 Synthetic Benchmark

Synchronization overhead for the `exploration` benchmark is shown in Figure 3. We tested 4 configurations per target: 1 producer and 1 consumer, and 1 producer and 4 consumers, 4 producers and 1 consumer, 2 producers and 2 consumers. All views of producer(s)/consumer(s) are connected to the same record structure with an horizon of 2^{24} elements. Each execution processes and communicates 2^{26} indices. Threads are pinned such that producer(s) are all mapped to different cores of the same chip, and all consumer are mapped to different cores on a *different* chip.

False sharing induces severe overheads for tiny bursts and is the cause of the wide performance instabilities. The burst size remains an important factor passed the cache line size, but synchronization overhead becomes negligible for bursts of 1024 indices or more. In addition, the scatter and gather performance is also excellent: small bursts remain profitable even for complex configurations with multiple producers and consumers. This validates the choice of an expressive concurrent data structure: performance is excellent on a simple pipeline, while offering maximal flexibility to support combinations of pipeline- and data-parallel computations in a high-level language.

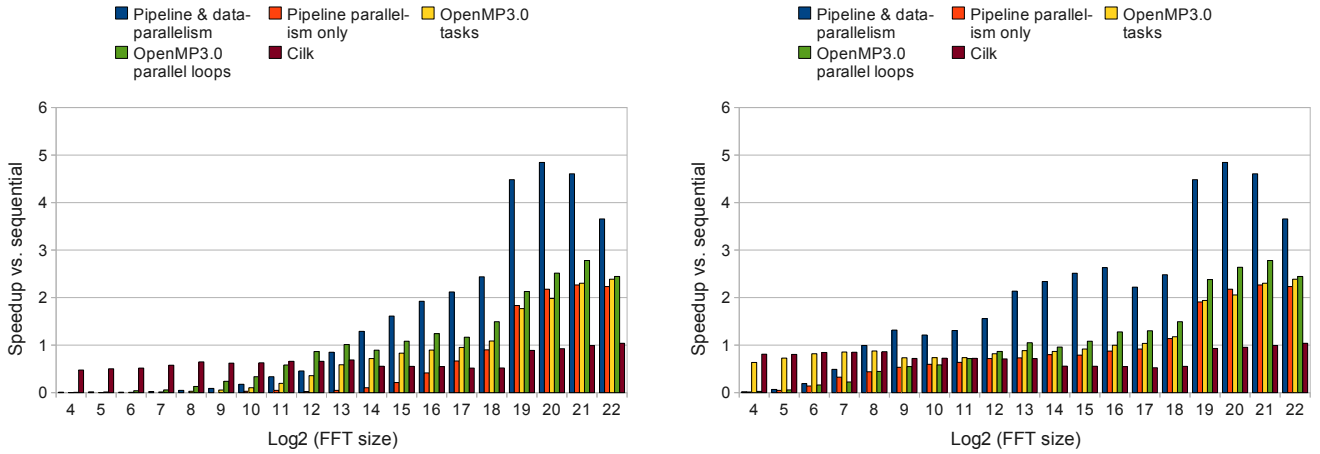The single-producer single-consumer configuration reaches a maximum of $103M$ index computations per second on

Figure 4: Performance of `fft` on Xeon. Single settings (left) and best settings per data point (right)
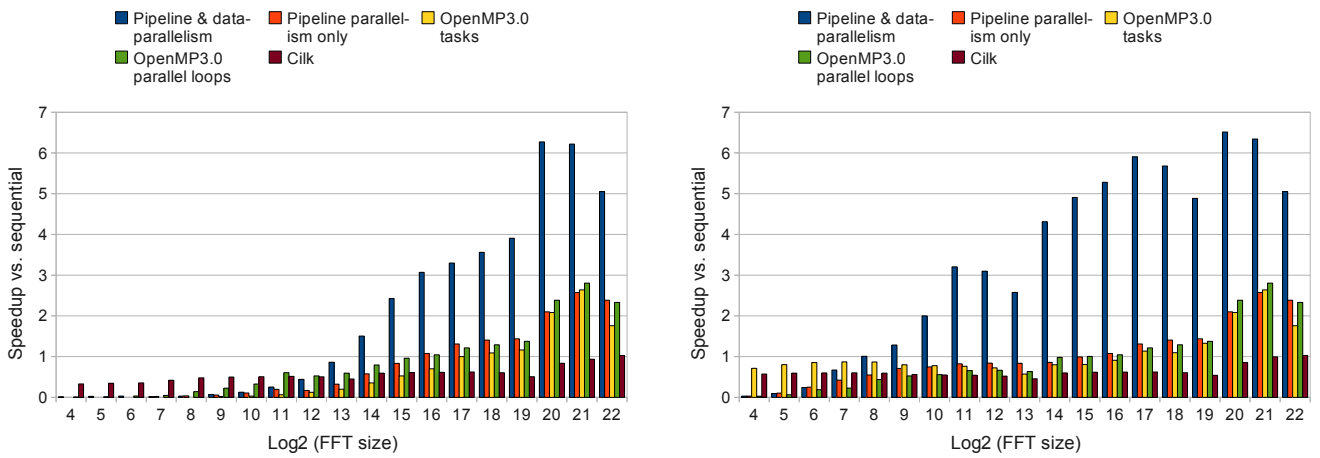

Figure 5: Performance of `fft` on Opteron. Single settings (left) and best settings per data point (right)

Xeon: on average 23.3 cycles per index, an order of magnitude shorter than the cache line transfer across x86 chips. This paradox is easily explained: (1) for a large-enough horizon, `update()` and `stall()` almost never result in a blocking synchronization, and (2) our cache-conscious algorithm amortizes cache line transfers over a large number of calls to the synchronization primitives.

5.2 Real Applications

Figures 4, 5 and 6 compare the performances of various parallel versions of the FFT kernel and considering multiple vector sizes. The baseline is an optimized sequential FFT implementation used as a baseline for the StreamIt benchmark suite. The first two bars are two parallel versions using ERBIUM, the next two bars are OpenMP versions, the last bar is a Cilk version. Combined pipeline and data-parallelism achieve the best speedups, compared to pure data-parallelism (both with ERBIUM). The size of the machines and the associated cost of inter-processor communication sets the break-even point around vectors of 256 single-precision floating point values.

FFT does not naturally expose much task parallelism because of the dependence patterns in the butterfly stages, yet
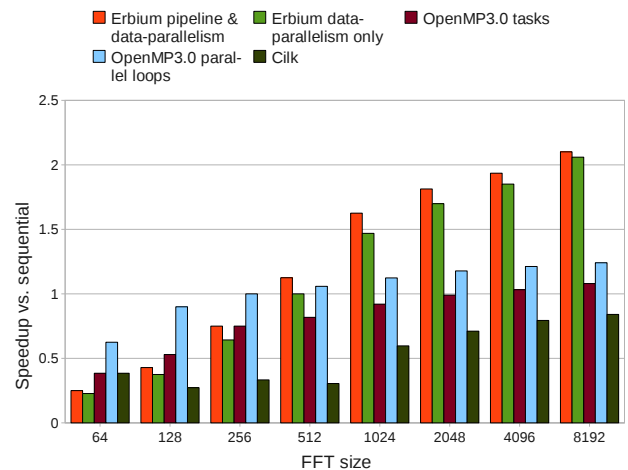

Figure 6: Performance of `fft` on Core 2

pipelining computations across different stages remains possible, and favors local, cache-to-cache communications over

external memory accesses; this explains the performance improvement of the combined version. On Xeon and Opteron exploiting this pipeline parallelism allows to reduce contention, even if at the expense of some data-parallelism. To better analyze the intrinsic synchronization performance of ERBIUM, we also show performance results on the smaller single-node Core 2 platform on Figure 6. As the possible concurrency is reduced, the data-parallel versions stand more of a chance, with a shared L3 cache and L2 shared among each two cores. However, even in this unfavorable setting, the addition of pipelining gives enough edge to our combined data- and pipeline-parallelism approach, which outperforms the task-parallel and data-parallel implementations in OpenMP, as well as a Cilk implementation [27].
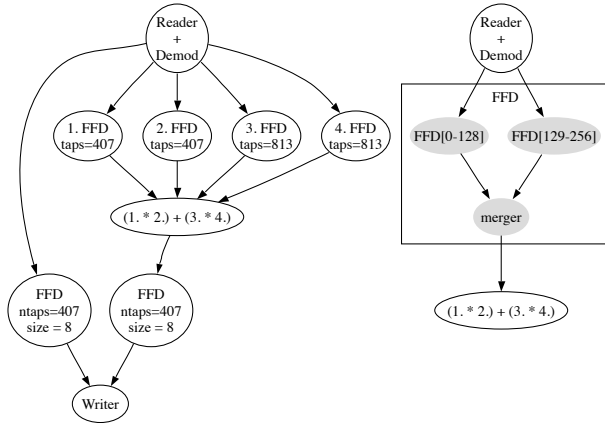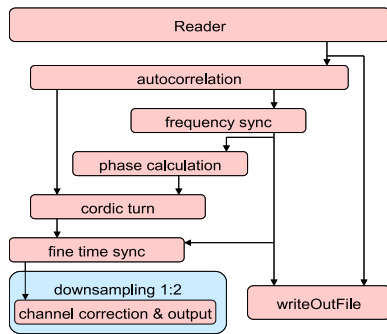

Figure 7: Informal data flow of fmradio

Figure 8: Informal data flow of 802.11a

Platform (cores)	Task-Level Only	Data-Parallel Only	Combined
Xeon (24)	1.85	1.84	6.67
Opteron (16)	2.73	2.81	7.45

Figure 9: Speedups results for 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 10: Speedups results for fmradio

Now considering the three full applications, fmradio and 802.11a did not require radical design changes to achieve scalable performance; jpeg was written initially as a fine-grain KPN and only required systematic conversion of the synchronization and communication methods.

On fmradio, exploiting task and pipeline parallelism is easy but shows limited scalability — 6 concurrent processes. Exploiting data parallelism is not trivial and involves an interesting transformation: the original code uses a circular window using modulo arithmetic and holding the results of previous filtering iterations; it can be replaced by a record, removing spurious memory-based dependences. Furthermore, the work must be distributed over independent workers then merged into a single output stream; to eliminate data copying overhead, the implementation leverages decoupled data access and synchronization, and the extended Kahn semantics where multiple workers deterministically produce data in exclusive index ranges.

Figure 7 illustrates the concurrency exposed in fmradio. On the left, 4 processes called FFD (Float input, Float output and Double taps) 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 10 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, GCC's automatic vectorizer being capable of aggressive loop restructuring in presence of concurrency primitives and view accesses.

Figure 8 illustrates the concurrency exposed in `802.11a`. The data-flow graph is more unbalanced than fmradio and it is not fork-join. `frequency_sync` and `fine_time_sync` are stateful: they need to be decoupled from the rest of the pipeline to enable data-parallelization [31]. Figure 10 displays speedup results. The *Combined* column shows the benefit in exploiting both task-level and data parallelism. Strict data parallelization even degrades performance on Xeon due to work-sharing overheads.
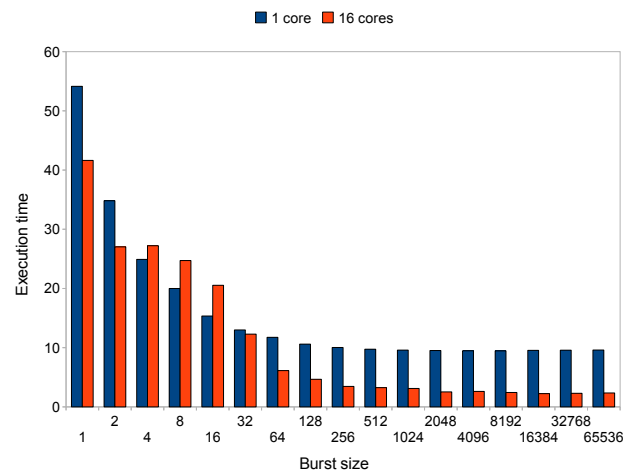

Figure 11: Performance of jpeg on Opteron

On jpeg, the systematic decomposition of the application exposes 23 computational tasks, communicating by exchanging burst of pixels (or coefficients, depending on the

filter/stage). In the uncompressed data stream, 64 pixels/-coefficients correspond to 1 macro block. The objective of this experiment is to demonstrate the benefits of ERBIUM on a real application with extremely fine grain tasks. Most of the fine-grain tasks can be further data-parallelized, but restrict ourselves to a task-parallel version for the purpose of this experiment. Figure 11 shows the results on Opteron, running the decoder once on a 4288×2848 image. Vertical axis is execution time in milliseconds, horizontal axis is the burst size in pixels. For single-core execution, the performance plateau is achieved for bursts of 128 pixels, or 2 macro blocks. For 16-core execution, the performance plateau is achieved for bursts of 512 pixels, or 8 macro blocks. Comparing the best 16-core and single-core versions, we achieve a $4.85\times$ speedup on Opteron. Most important, the break-even point — defined as the burst size where 16-core performance outperforms the best single-core performance — is *1 macro block only*. These numbers confirm that ERBIUM succeeds in exploiting fine-grain thread-parallelism on real applications, although better results could be achieved combining task-level and data parallelism. This is encouraging about the scalability on future manycore architectures, where data parallelism alone does not scale.

The tradeoff between task, pipeline and data-level parallelism depends on the target architecture, and is becoming one of the key challenges when adapting a computational application to a new platform. Our results show that ERBIUM is an ideal tool to explore this tradeoff. Overall, ERBIUM leverages much more flexible, scalable and efficient forms parallelism than restricted models.

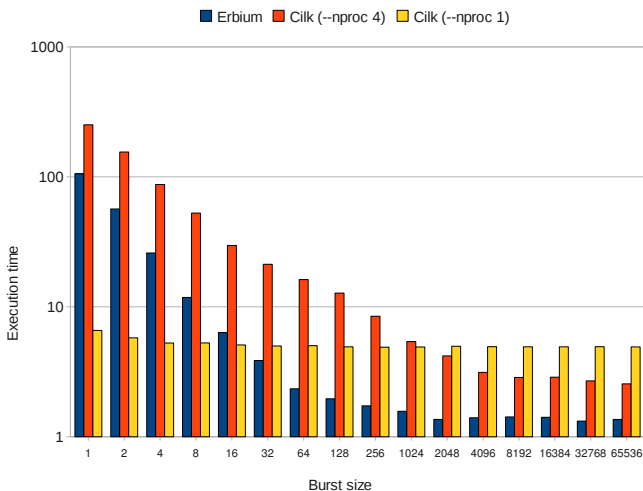## 5.3 Comparison with Lightweight Scheduling

Figure 12: Streaming processes vs. short-lived tasks

ERBIUM differs from the common parallel runtimes where concurrency is expressed at the level of atomic, short running tasks. Figure 12 compares the execution time of the `exploration` synthetic benchmark with a Cilk implementation spawning short-lived user-level tasks [27]. We consider the Core 2 target, 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 bursts sizes, and reaches almost $5\times$ 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 $80\times$ smaller than Cilk. It demonstrates that communications among long-lived processes are an essential abstraction for scalable concurrency.

These numbers also explain the poor performance of the Cilk FFT implementation. Data-parallelism dominates the scalability of the FFT, and Cilk incurs a noticeable scheduling and synchronization overhead for data-parallel execution. ERBIUM avoids this overhead by running data-parallel tasks fully independently, and implementing synchronizations across butterfly stages at a much lower cost.

We also compared ERBIUM with StarSs, in its SMPSs flavor [28]. StarSs is perfectly suited to express our data-flow applications. However, its current execution model relies on lightweight scheduling. Our experiments with `fmradio` show that StarSs achieves $3.88\times$ speedup on Xeon and $2.97\times$ on Opteron, 3 to 4 times less than ERBIUM.

Short-lived atomic tasks may be better supported with dedicated hardware [24]. This is also the case for streaming communications, as illustrated by the TTL approach [20]. Of course, lightweight threading techniques are still required for load balancing and to increase the reactivity of passive synchronizations (blocking `update()`/`stall()`).

6. RELATED WORK

Concurrency models have been designed for maximal expressiveness and generality [21,29], with language counterparts such as Occam [9]. Asynchronous versions have been proposed to simplify the implementation on distributed platforms and increase performance [14], with language counterparts such as JoCaml [13]. Compared to these very expressive concurrency models, ERBIUM builds on the Kahn principle (data flow), which is sufficient to expose scalable parallelism in a wide spectrum of applications; it also offers determinism and liveness guarantees that evade the more expressive models.

Our work is strongly influenced by the compilation of data-flow and streaming languages, including I-Structures [3], SISAL [23], Lustre [18], Lucid Synchrone [7], Jade [36] and StreamIt [41]. These languages share a common interest in determinism (time-independence) and abstraction. They also involve advanced compilation techniques, including static analysis to map declarative concurrent semantics to effective parallelism, task-level optimizations and static scheduling. ERBIUM is an ideal representation to implement platform-specific optimizations for such languages.

Extensions of OpenMP have been proposed to support pipeline parallelism and streaming applications [6,28,33,35,38]. These are promising tradeoffs between declarative abstractions and explicit, target-specific parallelization. But they suffer 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 $3\times$ on 4-core to 16-core x86-64 platforms [35]. A new proposal for a streaming extension of OpenMP has benefitted from our experience with

ERBIUM [34]; implementation is underway, with dedicated algorithms to convert the short-lived tasks of the OpenMP specification into persistent ERBIUM processes.

The work of Haid et al. [17] shares many design goals with ERBIUM. It aims for the scalable and efficient execution of Kahn process networks on multicore processors. Its sliding window design matches the layout of records in ERBIUM. But it does not come with an associated compiler intermediate representation, and it does not deal with modular composition and dynamic process creation. Furthermore, although it advocates for streaming communications, it still relies on dynamic scheduling for event-driven synchronization. Most encouraging to us, Haid et al. demonstrate that data-driven scheduling and streaming communications can coexist: it indicates that our approach is complementary to the large body of work in lightweight runtimes.

Fastflow is closely related with the ERBIUM runtime implementation [2]. It is a C++ programming pattern for streaming applications dedicated to shared-memory platforms. Its lock-free, fence-free synchronization layer has comparable performance to our single-producer single-consumer record. It supports multi-producer multi-consumer communication at the expense of an additional arbitration thread and memory copying. Because index-range negotiation is exposed to the compiler in the intermediate representation, our runtime achieves lock-free, fence-free multi-producer multi-consumer communication without these overheads.

Cilk does not natively support data-flow concurrency [27] but we share its emphasis on compile-time specialization. We wish to integrate work stealing and load-balancing capabilities in the future, as motivated by Azevedo et al. for irregular 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 (parallel data-flow annotations [32]) and StarPU (scheduling framework [4]).

Further performance improvements can be achieved with hardware support, starting from the pioneering data-flow architectures [11,42], recently revived for coarser-grain, task-level concurrency [1,24,37]. Streaming register files have also been proposed to accelerate data-flow computations [16,31].

Finally, our work is not related to the software approaches to achieve deterministic execution for debugging purposes — deterministic replay [30]. These approaches do not define a target-independent semantics for the application.

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 record 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.

8. 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] M. Aldinucci, M. Meneghin, and M. Torquati. Efficient Smith-Waterman on multi-core with FastFlow. In *Euromicro Intl. Conf. on Parallel, Distributed and Network-Based Processing*, pages 195–199, Pisa, Feb. 2010.

[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] 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.

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

[8] 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.

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

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

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

[12] H. M. et al. Acotes project: Advanced compiler technologies for embedded streaming. *Intl. J. of Parallel Programming*, 2010. Special issue on European HiPEAC network of excellence member's projects.

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

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

[15] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *ACM Symp. on Principles and practice of parallel programming (PPoPP'08)*, pages 43–52, Salt Lake City, Utah, 2008.

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

[17] W. Haid, L. Schor, K. Huang, I. Bacivarov, and L. Thiele. Efficient execution of Kahn process networks on multi-processor systems using protothreads and windowed FIFOs. In *Workshop on Embedded Systems for Real-Time Multimedia (ESTImedia'09)*, pages 35–44, Grenoble, France, Oct. 2009.

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

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

[20] T. Henriksson and P. van der Wolf. TTL hardware interface: A high-level interface for streaming multiprocessor architectures. In *Workshop on Embedded Systems for Real-Time Multimedia (ESTImedia'06)*, pages 107–112, Seoul, Korea, Oct. 2006.

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

[22] 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.

[23] 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.

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

[25] 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.

[26] E. A. Lee and A. L. Sangiovanni-Vincentelli. A framework for comparing models of computation.

[27] 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.

[28] V. Marjanovic, J. Labarta, E. Ayguadé, and M. Valero. Effective communication and computation overlap with hybrid MPI/SMPSs. In *PPOPP*, 2010.

[29] 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.

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

[31] 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.

[32] 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.

[33] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with starss. *Intl. J. on High Performance Computing Architecture*, 23(3):284–299, 2009.

[34] A. Pop and A. Cohen. A stream-comptuing extension to OpenMP. In *Proc. of the 4th Intl. Conf. on High Performance and Embedded Architectures and Compilers (HiPEAC'11)*, Jan. 2011.

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

[36] 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.

[37] 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.

[38] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou, and P. Trancoso. Tflux: A portable platform for data-driven multithreading on commodity multicore systems. In *Intl. Conf. on Parallel Processing (ICPP'08)*, pages 25–34, Portland, Oregon, Sept. 2008.

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

[40] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'10)*, Vienna, Austria, Sept. 2010.

[41] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In

Intl. Conf. on Compiler Construction, Grenoble, France, Apr. 2002.

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