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

Cupertino Miranda[1] Philippe Dumont[1,2] Albert Cohen[1]
Marc Duranton[2] 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

ABSTRACT

Optimizing compilers and runtime libraries do not shield programmers from the complexity of multi-core hardware; as a result the need for manual, target-specific optimizations increases with every processor generation. High-level 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.

This paper introduces ERBIUM, an intermediate representation for compilers, a low-level language for efficiency programmers, and a lightweight runtime implementation. It relies on a data structure for scalable and deterministic concurrency, called *Event Recording*, exposing the data-level, task and pipeline parallelism suitable to a given target. We provide experimental evidence of the productivity, scalability and efficiency advantages of ERBIUM, relying on a prototype implementation in GCC 4.3.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features; D.3.4 [Programming Languages]: Processors - Run-time environments, Compilers

General Terms: Algorithms, Languages, Performance

1. DESIGN AND SEMANTICS

ERBIUM defines an intermediate representation for compilers, also usable as a low-level language by efficiency programmers. Our objectives are the following.

- **Determinism.** ERBIUM's semantics derives from *Kahn Process Networks* (KPNs) [5]. KPNs are canonical concurrent extensions of (sequential) recursive functions preserving *determinism* (time independence) and *functional composition.* 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.
- **Expressiveness.** In concurrent data-flow languages, data and functional parallelism is implicit in (recursive) functions. As an intermediate representation, ERBIUM provides explicit, asynchronous spawn points for concurrent processes. Unlike periodic subclasses of KPN [1], 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 [2,3,6] or specific parallel computation skeleton [8]. It supports program transformations for dynamic, data-dependent control flow applications, including generalized forms of decoupled software pipelining [4,7]. The compiler is responsible for selecting an appropriate specialization, offering the most relevant runtime primitives and interface for a given platform.
- **Lightweight implementation.** ERBIUM is designed to be as close as possible to the hardware while preserving portability and determinism. Any intrinsic overhead in its design and any implementation overhead will hit scalability and performance; such overheads cannot be recovered by programmers who operate at this or higher levels of abstraction.

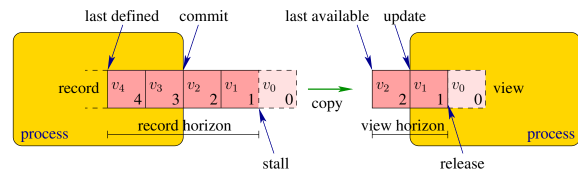

Figure 1: Data flow with bounded resources

Figure 1 illustrates the ERBIUM primitives and event recording structures on a simple producer-consumer template. The `commit()` and `update()` primitives implement data-flow *pressure*, enforcing causality among processes. In this split-phase design, data-flow communication is decoupled from

Copyright is held by the author/owner(s).
CF'10, May 17–19, 2010, Bertinoro, Italy.
ACM 978-1-4503-0044-5/10/05.

```
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

synchronization. A one-sided, asynchronous communication is initiated with the **receive()** primitive. The **release()** and **stall()** primitives implement *back-pressure*. 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). The stall index is always lower than or equal to the minimum of the connected views' release indices.

Figure 2 shows an illustrative producer-consumer example. A single recording is connected to a single view. Dataflow synchronization, communication and back-pressure are straightforward. Each process sets its own recording/view horizons and its own commit/update bursts. In a given burst, **commit()** follows the last definition of a value, **update()** precedes the first use, **release()** follows the last use, and **stall()** precedes the first definition.

Termination detection by the consumer takes two phases: the value returned by **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 descriptor owned (allocated) by the producer is an initialization argument for the consumer; it has been initialized prior to spawning the producer, and is used to connect the view to the recording in the consumer. Together with the backpressure design, it makes separate compilation of the producer and consumer possible.

2. EXPERIMENTS

Our experiments target real applications and a code generator implemented in an experimental branch of GCC 4.3. The code generator expands the ERBIUM constructs to their shared-memory specializations *after* the main optimization passes. The most interesting step is to hide the concurrency constructs from the compiler optimizations. This peaceful collaboration of thread-level parallelism with middle- or back-end optimizations is the result of our intermediate language approach, and is rarely found in high-level languages or low-level threading libraries.

Platform (cores)	fmradio	802.11a	jpeg
Xeon (24)	12.6	6.67	2.42
Opteron (16)	14.6	7.45	1.95

Figure 3: Speedups for fmradio, 802.11a and jpeg

We report experimental data about the parallelized version of our three applications. **fmradio** (GNU radio) and **802.11a** (internal reference code at Nokia) did not require algorithmic or radical design changes to achieve scalable performance. In the case of **jpeg**, we chose to illustrate the expressiveness and low-overhead benefits of ERBIUM: the code was initially decomposed at the finest possible grain exposing KPN semantics, and we did attempt to hide synchronization latency through task coarsening or fusion. Figure 3 summarizes the speedups on 24-core and 16-core x86 platforms. The baseline is the sequential (original) version compiled with -O2. Streaming codes are often bandwidth-bound: the Xeon's front-side bus appears to be penalized on such codes compared to the Opteron's Hypertransport busses: data-parallelism is limited by off-chip memory bandwidth. It is encouraging that **jpeg** shows a modest but real speedup despite its unrealistically fine grain parallelism.

3. 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. ERBIUM is implemented in GCC 4.3, allowing classical optimizations and parallelizing transformations to operate transparently. It relies on 4 concurrency primitives implemented with platform-specific, non-blocking algorithms. 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 porting ERBIUM to distributed memory machines (Cell BE and clusters), and on front-ends for high-level, performance portable languages.

4. REFERENCES

[1] 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.

[2] 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.

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.