Session-Based Compilation Framework for Multicore Programming

Nobuko Yoshida[1], Vasco Vasconcelos[2], Hervé Paulino[3], and Kohei Honda[4]

[1] Department of Computing, Imperial College London
[2] Lasige, Department of Computer Science, University of Lisbon
[3] CITI, Departamento de Informática, Universidade Nova de Lisboa
[4] Department of Computer Science, Queen Mary, University of London

Abstract. This paper outlines a general picture of our ongoing work under EU Mobius and Sensoria projects on a type-based compilation and execution framework for a class of multicore CPUs. Our focus is to harness the power of concurrency and asynchrony in one of the major forms of multicore CPUs based on distributed, non-coherent memory, through the use of type-directed compilation. The key idea is to regard explicit asynchronous data transfer among local caches as typed communication among processes. By typing imperative processes with a variant of session types, we obtain both type-safe and efficient compilation into processes distributed over multiple cores with local memories.

1 Introduction

This paper presents a brief overview of our ongoing work under EU Mobius and Sensoria projects on a type-based compilation and execution framework for distributed-memory multicore CPUs. Our aim is to obtain a new level of understanding on the effective shape of compilation and runtime architecture for distributed-memory chip-level multiprocessing. We take the viewpoint that communication and concurrency are a natural and fundamental structuring principle for modern applications. We identify typed processes exchanging messages through asynchronous communication as a basic model of computation, which we reify as a typed intermediate language. This intermediate language acts both as the target of translation from high-level programming languages and as the source of compilation to distributed memory chip-level multiprocessors. In both translation processes, types for communicating processes are used for ensuring key correctness properties for the resulting low-level code.

The background of this project is a recent fundamental change in the internal environment of computing machinery, driven by limiting physical parameters in VLSI manufacturing process [13, 34, 36], from monolithic Von Neumann architectures to chip-level multiprocessing (CMP), or CPUs with multiple cores. In the present work we are mainly interested in the CMP architectures based on distributed memory [24, 35], which offer the hardware interface analogous to distributed memory parallel computers [8] (in contrast to SMP/ccNUMA-like cache coherent CMP architectures [25, 27, 44]). This choice reflects our belief that a major factor for maximally exploiting the physical

potential of future microprocessors is how one can harness asynchrony and latency in intra-chip data transfer.

A non-uniform access to memories inside a chip can be realised by different methods, such as cache-line locking, eviction hints and pre-fetching. One method, often used for distributed memory CMP, employs direct asynchronous memory-to-memory data transfer, or Direct Memory Access (DMA), to share data among cores' on-chip local memories. A central observation underlying this approach is that trying to annihilate distance (i.e. to maintain global coherence) can be too costly, just as maintaining hardware interface for coherent distributed shared memory over a large number of nodes is unfeasible. This observation favours the use of explicit operations for directly transferring data from one part of a chip to another, and one of the efficient methods for doing so, effectively exploiting intra-chip communication bandwidth, is DMA operations. In a high-level view, this approach regards CMP as distributed parallel machines with explicit remote data transfer among them, making the framework close to computing models such as the LogP model [7] and parallel hierarchical memories [1]. The direct, asynchronous memory-to-memory transfer as a means of data exchange is flexible and can potentially make the most of on-chip network bandwidth [26], which is many-fold larger than intra-host computer networks [9], promoting concurrent, asynchronous use of communication and computing elements inside a chip. As has been studied in the literature [15–17, 30, 31], message passing concurrency can flexibly and generally represent the diverse forms of control and data flows found in sequential and concurrent applications. At the same time, the very nature of DMA operations, in particular asynchronous, direct rewrite of local memory of a distributed core, makes it hard to harness their power with safety assurance and controllability comparable to the traditional sequential hardware (for further discussions on this model, see §2.1).

In future, high-level applications will be designed and programmed using many different abstractions, especially regarding concurrency [6, 28, 39, 40, 42]. To understand the programming potential of distributed memory CMP, we need to examine whether these diverse abstractions, with associated data and control flow, can be mapped to this hardware model with efficiency, precision and fundamental safety assurance. One of the central concerns in this regard is to find an effective, disciplined method for using the DMA operations, making the most of their raw, asynchronous nature for flexibility and expressiveness while ensuring their correct usage. The desirable correctness properties include the freedom from synchronisation and racing errors (in the sense that data is remotely written only when a receiver is expecting it and at an expected region, and no other simultaneous writes can corrupt the data), the freedom from type errors (only data of a expected type and size is written), and progress of ongoing conversations (interaction sequences take place following the expected structure: in particular, a receiver will eventually obtain an expected datum).

In this paper we discuss one approach to the general compilation framework for distributed memory CMP. The framework is intended to offer a general, uniform and flexible basis for realising efficient translations of diverse (concurrent) programming abstractions to CMP executable code, with a formal guarantee of the aforementioned key correctness properties. The basic idea of our approach is to stipulate *typed communicating processes* as a representation for an intermediate compilation step from high-

level abstractions, and, after a type-based analysis of this intermediate representation, perform a *type-directed compilation* [32] onto executable binary code for distributed memory CMP. Schematically:

High-level concurrent languages (L2)
⇓
Typed imperative processes (L1)
⇓
CMP executable (L0)

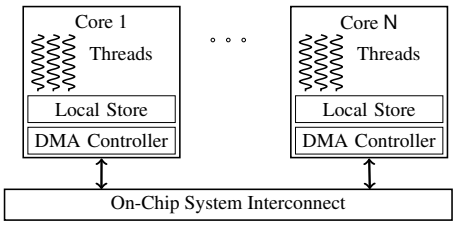
Above L0, L1, L2 refer to abstraction levels. Each ⇓ stands for one or more type-preserving compilations. At L1, we use an intermediate concurrent imperative language with types for channel-based conversations. The preceding studies on types for communicating processes, many centring on the π-calculus, have shown that they can offer fundamental articulation and basic safety guarantee for diverse communication patterns. As communication types for the compilation framework, we use a variant of session types [19, 41] for multiparty interactions [3, 5, 20], into which various high-level abstractions can be translated and which allows their efficient and safety-preserving compilation to distributed CMP primitives. The session types at L1 are generated from the interaction structures implicit in the high-level abstractions in L2, as we shall illustrate with a concrete example in the subsequent sections. The resulting typed communicating processes are amenable to uniform program analyses for safety assurance, and can be directly mapped to efficient code in L0, with a formal guarantee of the aforementioned key correctness properties.

2 Preliminaries

In this section we first clarify our assumptions on a hardware model, followed by a brief illustration of essential features of DMA operations. Then we present a running example for our type-preserving compilation framework, a simple streaming application. In particular we focus on the behaviour of the *double-buffering algorithm* used for compiling the running example. The algorithm is the standard method for stream and media processing to make the best of high-performance, multicore computing [21, 37].

2.1 A Hardware Model and DMA Primitives

Hardware Model. We assume an idealised model where a chip consists of multiple cores of the same Instruction Set Architecture (ISA), each with a local memory. Cores may or may not allow preemptive threads. Data sharing among distributed cores is performed via asynchronous data transfer from one local memory to another (DMA), as illustrated in the following diagram.



Our focus in the present inquiry is on the DMA-based data sharing among distributed memories: we do *not* consider other issues in distributed memory CMP such as the size of local memory, hierarchical memory organisation, capability control, security and heterogeneity. These are relatively orthogonal issues whose analysis may benefit from the understanding of the factor studied in the present paper.

DMA Primitives. Two versions of DMA primitives are known, an asynchronous write ("put"), and an asynchronous read ("get"). We mainly focus on *put* for brevity. The semantics of the *put* does not demand the sender to know the arrival of data for its sending action to complete: it is a non-blocking write. This asynchronous nature is essential for efficiency. Since a remote operation is anyway relatively expensive (even inside a chip [26]), we amortise the cost by sending a block or blocks of words, which can total hundreds of thousands of bytes. A sender can block until the data is sent out, or can be asynchronously notified. The DMA gains further efficiency by sending (even contiguous) words out-of-order. The receiver can be notified either asynchronously by a different messaging/interrupt mechanism or by a subsequent locally ordered *put* to an associated flag: for example one can place a memory fence [23] between the first *put* for data transfer and the subsequent *put* for a flag, so that the write to the flag (say turning 0 to 1) takes place *after* all the writes for the first *put*. Since consecutive writes are often cheap, this is an efficient method for checking the delivery.

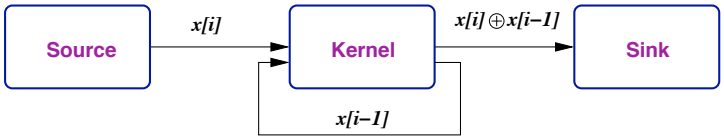
Throughout the present paper, we assume a "macro" command for *put*, which includes initiating a send operation (including, if we use the scheme discussed above, a subsequent fenced flag) and waiting for the data to be sent out from the sender's local memory, but *not* for its arrival (and writing) at the receiver's remote memory. Thus, as soon as the data has been sent out, the CPU will become free. This is the standard usage of *put* [26], based on which we can easily accommodate an asynchronous notification as simple optimisation. Dually we assume a single macro command *wait* for the receiving side of *put*, which can, for example, consist of waiting for a fenced flag to be turned from 0 to 1, as discussed above. Each of these macros can be realised by a few hardware instructions [23], with different schemes depending on the mechanisms offered by a given hardware/software environment.

Observations on DMA Primitives Because of its efficiency and flexibility, DMA is often used (partially or wholly) in multiprocessor system-on-chips. One of the prominent recent examples include Cell microprocessor [35]. This model considers CMP as a microscopic form of distributed computing, and is capable of making the most of on-chip interconnect, suggesting its potential scalability when the number of cores per chip increases and a relative wire delay inside a chip takes effect [9]. It can realise arbitrary

forms of data sharing among cores' local memories, and in that sense it is general-purpose. Being efficient and general-purpose, however, the DMA operations are also extremely hard and unsafe to program in their raw form: the very element that makes the DMA operations fast and flexible — asynchronous, direct rewrites of memory — also makes them unwieldy and dangerous. The direct writes of one memory area to another, asynchronously issued and asynchronously performed, can easily destroy the works being conducted by applications. The danger of programming using these asynchronous operations may be compared to that of bare machine-level programming in sequential computers, without assistance of high-level language constructs such as procedures and data types and the associated compilation infrastructure, aggravated by the presence of concurrency and asynchrony.

2.2 Stream Processing and Double-Buffering

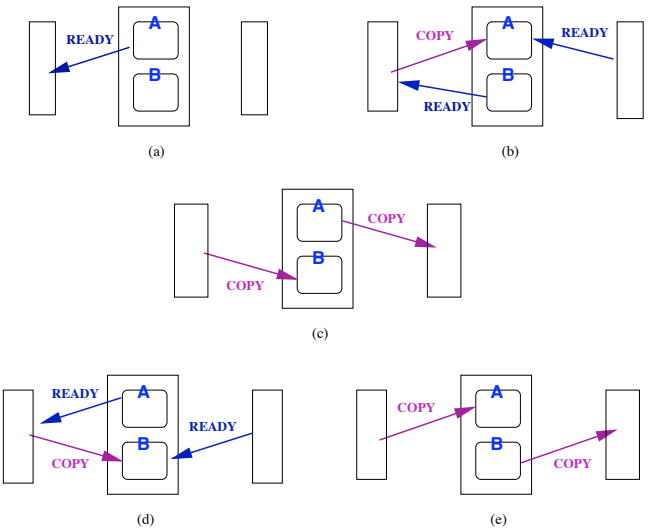
A Simple Stream Program We take a simple stream program for data encryption as an illustration of our compilation framework [38]. Consider the following stream graph.



A data producer *Source* continuously feeds data to a *Kernel*, which calculates the XOR of each element with a key and writes the result on a stream to a consumer *Sink*. *Sink* may also have its own processing on the resulting data. The key used at each turn comes from (except for the first time) the Kernel's own output through a feedback, for a cipher block chaining. Such a stream algorithm can be easily expressed in stream programming languages [4, 12, 39, 43], whose program consists of transformers (called *kernels* or *actors*) connected through directed streams: each actor gets data from its incoming streams, processes them and places the results to its outgoing streams. For example, in the application above, a stream program for Kernel will be specified as a transformer which receives (say) an integer x from an incoming stream, calculates the XOR $x \oplus p$ where p is a variable storing the preceding value of x (where the initial value of p would be set to be some encryption key), and places the resulting value to an outgoing stream as well as assigning it to the new value for p. Stream programming has applications in DSP, multimedia and scientific computing and enables natural exploitation of parallelism at various levels, starting from high-level transformation of stream graphs to DMA-based multicore execution.

Double Buffering In order to execute such a stream graph in a distributed memory CMP, the first step is to enlarge data granules, through the standard strip mining technique [29]: for example we may decide to treat these streams by units of say 16kB. This allows actors to exchange data in large blocks, instead of byte by byte (which would incur high overheads). We then program these three actors to exchange data strips (each of size 16kB) through an interactional algorithm called *double buffering* [37], illustrated

Fig. 1 Double-Buffering



in Figure 1, which is often found at the heart of implementations of stream programs in CMP. [5] Kernel uses two 16kB arrays, or *buffers*, named A and B in the picture: while Source uses a single 16k array (in practice it can use a large cyclic buffer), fed by, say, a byte stream from an external channel. The central idea of the algorithm is to repeat the following procedure.

> *While Kernel is receiving data into array* A *from Source, it processes data in array* B *and sends the result to Sink; it then repeats the process by exchanging the roles of* A *and* B.

The five steps in Figure 1 materialise this idea:

(a) Kernel tells Source it is ready to receive an initial strip at buffer A;
(b) Source starts sending to A; asynchronously Kernel tells Source it is *also* ready to receive at buffer B, and again asynchronously Sink tells Kernel it is ready to receive at its own 16kB array;
(c) Kernel finishes processing its A-strip and sends the resulting data to Sink, while Source is sending a strip to B;
(d) Source continues sending to B; Kernel asynchronously tells Source it is ready to receive at A (since Kernel has now sent out its A-strip); again asynchronously Sink tells Kernel it is ready to receive the next strip;
(e) Now the situation is symmetric with respect to **(c)**: Source writes to A and Kernel writes from B. We now go back to **(b)**.

[5] An effective method to allocate/schedule actors in a CMP environment is an interesting problem: we do not address this issue here because it involves runtime resource management, which is outside the focus of our present discussions.

The algorithm allows asynchrony among computations and communications with minimal synchronisation to prevent data pollution. By overlapping computation and communication [7, 14], it makes the most of the available hardware resources, allowing concurrent and asynchronous execution of computation and communication. This allows the effective usage of available communication bandwidth in code execution, the tenet of effective network programming.

3 The Intermediate Language with Multiparty Session Types

This section introduces imperative processes with multiparty session types [41] as an intermediate language using the double-buffering example. This intermediate language serves two purposes. First, it provides an effective *source* language for compilation into a typed assembly language for CMP. Second, it offers an expressive *target* language into which we can efficiently and flexibly translate different kinds of high-level programs. This latter aspect is based on the observation that many concurrent and potentially concurrent programs (such as a streaming example above) can be represented as a collection of structured conversations, where we can abstract the structure of data movement in their programs as types for conversations. Through the use of these types and associated program analyses, we can formally ensure communications in programs are free from synchronisation and type errors, and satisfy progress.

3.1 Double Buffering in the Intermediate Language

The double buffering algorithm is both imperative and interactional, with highly structured communication structures. Asynchrony between sending and receiving is fundamental for its efficiency. The aim of the design of the intermediate language `L1` (for *Session-typed Intermediate Language*) is to allow a precise and flexible typed description of such interactional imperative programs with precision and flexibility, in a form directly translatable to the execution mechanisms of distributed CMP.

Figure 2 shows the description of the double buffering algorithm in `L1`. Program Main first finds three idle (virtual) cores denoted $\mathtt{p}_0$ to $\mathtt{p}_2$ (`newPlace`) and creates a new service channel a (`newChan`) to be used for the session initialisation by programs Source, Kernel, and Sink, running at different cores (`spawn`). Each "place" denotes an abstract unit of processing and memory resources, which may as well be considered as a virtual notion of a core with local memory in a distributed CMP chip.

The first line in Source represents session initialisation (on channel a); this is the point where Source receives the channels shared by all participants. The asynchronous session types in `L1` require distinct channels to be used for distinct communications (except for communications iterated in a loop, which use the same channels), essential for translation into DMA operations. Thus we use four channels r_1, r_2, s_1, s_2 between Source and Kernel, and another four between Kernel and Sink. Now Source starts a conversation: after feeding its array through a for-loop ($\mathtt{foreach}(i : 1..n)$ denotes the *pointwise iterator* for processing arrays which allows us to work with non-trivial programs without addressing array-bound checks [6]), it waits for an "A-ready" signal through r_1 ("?" denotes input), sends the data in the array through s_1 ("!" for output);

Fig. 2 Double-Buffering Algorithm in L1

Main Program :

```
newPlace p0, p1, p2;
newChan a;
spawn(Source(a))@p0;
spawn(Kernel(a))@p1;
spawn(Sink(a))@p2
```

Source(a) :

```
a[0](r1 r2 s1 s2 t1 t2 u1 u2).
newVar y : int[n];
μX.(
  //send to x_A
  foreach(i : 1..n){
    y[i] = get_int();
  }
  r1?(); s1!⟨y⟩;
  //send to x_B
  foreach(i : 1..n){
    y[i] = get_int();
  }
  r2?() ;s2!⟨y⟩;X
)
```

Kernel(a) :

```
a[1](r1 r2 s1 s2 t1 t2 u1 u2).
newVar x_A, x_B : int[n];
newVar key : int = KEY;
r1!⟨⟩; r2!⟨⟩;
μX.(
  //process x_A
  s1?x_A;
  foreach(i : 1..n){
    x_A[i] := x_A[i] ⊕ key;
    key := x_A[i];
  };
  t1?(); u1!⟨x_A⟩; r1!⟨⟩;
  //process x_B
  s2?x_B;
  foreach(i : 1..n){
    x_B[i] := x_B[i] ⊕ key;
    key := x_B[i];
  };
  t2?(); u2!⟨x_B⟩; r2!⟨⟩; X
)
```

Sink(a) :

```
a[2](r1 r2 s1 s2 t1 t2 u1 u2).
newVar z : int[n];
μX.(
  //receive & print x_A
  t1!⟨⟩; u1?z;
  foreach(i : 1..n){
    print z[i];
  };
  //receive & print x_B
  t2!⟨⟩; u2?z;
  foreach(i : 1..n){
    print z[i];
  };
  X
)
```

repeats the same for r_2 and s_2, and returns to the main loop. Communication is purely asynchronous—the sending order is not guaranteed to be preserved at arrival.

Kernel, after allocating its variables (including the initial key value), signals Source that its buffers are both empty, via channels r_1 and r_2; then enters the main loop, where it proceeds as follows: first receives a datum at buffer x_A via s_1, goes through the buffer taking the XOR element-wise, after which it waits for Sink's cue via t_1 (which may have already arrived asynchronously), and finally sends out the buffer contents to Sink via u_1, and tells Source via r_1 that it is ready to receive at buffer A. It then works similarly for the second buffer. Sink acts in a way symmetric to Source (`print` prints a datum).

The three programs precisely describe the interactional behaviour informally illustrated in Figure 1.

3.2 L1 with Multiparty Session Types

We now outline how these structured dialogues can be abstracted as *types for conversations* in the form of multiparty session types [20], where pure asynchrony in communication is captured by a subtyping relation [33]. For example, in Figure 2, we see Source interacting with Kernel through channels r_1, s_1, r_2 and s_2 in this order, which is *different* from what we read from Kernel which starts by interacting at r_1 and r_2. How can we make sure that the Source's behaviour correctly matches that of the Kernel?

The theory of multiparty session types can type-abstract and verify the structure of a whole conversation. In the present context, the most notable feature of these types is that they can formally guarantee communication-safety and progress (deadlock-freedom). From a design viewpoint, developing a distributed program including a compilation framework demands a clear formal design as to how multiple participants communicate and synchronise with each other. These are the reasons why we start from a *global type* G, which plays the role of a type signature for distributed communications. These global types present an abstract high-level description of the protocol that all participants have to honour when an actual conversation takes place [20].

Once this signature G is agreed upon by all parties as the global protocol to be followed, a local protocol from each party's viewpoint, *local type* T_i, is generated as a projection of G to each party. If the global signature is too rigid, an individual party might wish to change their implementation locally. In this case, each local type T_i can be *locally refined* to, say, T_i', possibly yielding *optimised* protocols that are realised by programs P_i. If all the resulting local programs $P_1,..,P_n$ can be type-checked against refined $T_1',..,T_n'$, then they are automatically guaranteed to interact properly, without incurring in communication mismatch or getting stuck inside sessions, while precisely following the intended scenario.

Global Types. The development of type-safe programs for a double-buffering algorithm starts from designing the global type G,

$$
\begin{array}{l}
\mu\mathbf{t}.(\\
\mathsf{Kernel} \to \mathsf{Source} : r_1\langle\rangle; \quad \mathsf{Kernel} \to \mathsf{Source} : r_2\langle\rangle; \\
\mathsf{Source} \to \mathsf{Kernel} : s_1\langle U\rangle; \quad \mathsf{Source} \to \mathsf{Kernel} : s_2\langle U\rangle; \\
\mathsf{Sink} \to \mathsf{Kernel} : \quad t_1\langle\rangle; \quad \mathsf{Sink} \to \mathsf{Kernel} : \quad t_2\langle\rangle; \\
\mathsf{Kernel} \to \mathsf{Sink} : \quad u_1\langle U\rangle; \quad \mathsf{Kernel} \to \mathsf{Sink} : \quad u_2\langle U\rangle; \mathbf{t})
\end{array}
$$

where Source, Kernel and Sink denote participant names, identified as $\mathtt{p}_0, \mathtt{p}_1$ and $\mathtt{p}_2$ in the program in Figure 2.

A global type $\mathtt{p} \to \mathtt{p}' : k\langle U\rangle; G'$ means that participant $\mathtt{p}$ sends participant $\mathtt{p}'$ a message of type U on channel k, and then interactions described in G' take place. In this example, U denotes an int-array type. Type $\mu\mathbf{t}.G$ is use for recursive protocols where $\mathbf{t}, \mathbf{t}', \ldots$ are type variables.

The global type G uses recursion to describe an infinite loop where Kernel first notifies Source via r_1, r_2 that it is ready to receive data in its two channels s_1, s_2 (a signal at r_i says s_i is ready); Source complies, sending two chunks of data sequentially via s_1, s_2. Then Kernel (internally processes data and) waits for Sink to inform (via t_1, t_2) that Sink is ready to receive data via u_1, u_2; upon receiving the signals, Kernel sends the two chunks of processed data to Sink. This global protocol specifies a safe and deadlock-free scenario.

Local Session Types and Refinement. Once given global types, a programmer can develop code, one for each participant, incrementally validating its conformance to the projection of G onto each participant by efficient type-checking. When programs are executed, their interactions are guaranteed to follow the stipulated scenario. The type specification also serves as a basis for maintenance and upgrade.

Local session types abstract sessions from each endpoint's view. For example, Type $k!\langle U\rangle$ expresses the sending of a value of type U on channel k. Type $k?\langle U\rangle$ is its dual input. The relation between global and local types is formalised by *projection*, written $G \upharpoonright \mathtt{p}$ and called *projection of G onto* $\mathtt{p}$, defined as in [20].

Now we give the local types of Source, Kernel and Sink.

$$\begin{aligned} T_{\text{source}} &= \mu\mathbf{t}.r_1?\langle\rangle;s_1!\langle U\rangle;r_2?\langle\rangle;s_2!\langle U\rangle;\mathbf{t} \\ T_{\text{kernel}} &= \mu\mathbf{t}.r_1!\langle\rangle;s_1?\langle U\rangle;t_1?\langle\rangle;u_1!\langle U\rangle;r_2!\langle\rangle;s_2?\langle U\rangle;t_2?\langle\rangle;u_2!\langle U\rangle;\mathbf{t} \\ T_{\text{sink}} &= \mu\mathbf{t}.t_1!\langle\rangle;u_1?\langle U\rangle;t_2!\langle\rangle;u_2?\langle U\rangle;\mathbf{t} \end{aligned}$$

The local type of the program Kernel in Figure 2 is given below but it does not match the local type T_{kernel}, which is directly projected from global type G.

$$T^\star = r_1!\langle\rangle;r_2!\langle\rangle;\mu\mathbf{t}.s_1?\langle U\rangle;t_1?\langle\rangle;u_1!\langle U\rangle;r_1!\langle\rangle;s_2?\langle U\rangle;t_2?\langle\rangle;u_2!\langle U\rangle;r_2!\langle\rangle;\mathbf{t}$$

Our purpose is to refine T_{kernel} so that the new local protocol allows further asynchrony by overlapping communication and computation while still conforming to G [7, 14]; this allows us to start from a sequential global type, which is easily checked to be correct and deadlock-free, and refine it to a more optimised protocol, while guaranteeing that all participants still safely interact, e.g., that Kernel can interact with Source and Sink safely so that their interactions as a whole conform to the original global type G.

In the refined protocol $T^\star$, Kernel notifies Source via both r_1, r_2 *before* entering the loop, allowing Source to start its work. Now inside the loop, the refined protocol dictates that Kernel first receives data from Source via its first channel s_1, processes the data and sends out the result to Sink via its first channel u_1 and *immediately notifies Source via r_1 that it is ready on its first channel*, allowing Source to start sending data early. Kernel then repeats the same procedure for its second set of channels shared with Source and Sink. In this way, the refined local type says that Kernel can process data it has already received in one channel while still receiving data in the other, noting that sending, transferring and receiving large pieces of data can be time consuming.

We now summarise how this optimised local protocol is in fact safe with respect to the other participants conforming to G, through the notion of asynchronous communication subtyping. The justification is non-trivial: it uses a combination of a partial commutativity of the input and output actions and nested unfolding of recursive types [33]. The two key subtyping rules for permuting finite actions we use are as follows:

$$\begin{aligned} k!\langle U\rangle;k'?\langle U'\rangle;T_0 &\ll k'?\langle U'\rangle;k!\langle U\rangle;T_0 \qquad (k \neq k') \\ k!\langle U\rangle;k'!\langle U'\rangle;T_0 &\ll k'!\langle U'\rangle;k!\langle U\rangle;T_0 \qquad (k \neq k') \end{aligned}$$

In the first rule, the left-hand type allows for more asynchrony (optimal) than the right-hand side type since the output action on k can be performed without waiting for the input on k'. The second rule permutes the two outputs at distinct names since they are sent asynchronously. The rule $\ll$ are applied to only finite length of the session types (hence $\ll$ is decidable). We write $T \gg T'$ for $T' \ll T$.

To define the subtyping for recursive types, we need to combine $\ll$ with unfolding. We call a relation $\mathfrak{R} \in \mathit{Type} \times \mathit{Type}$ an *asynchronous subtype simulation* if $(T_1, T_2) \in \mathfrak{R}$ implies the following conditions.

1. If $T_1 = \mathsf{end}$, then $\mathsf{unfold}^n(T_2) = \mathsf{end}$.
2. If $T_1 = k!\langle U_1\rangle;T_1'$, then $\mathsf{unfold}^n(T_2) \gg k!\langle U_2\rangle;T_2'$, $(T_1', T_2') \in \mathfrak{R}$ and $(U_1, U_2) \in \mathfrak{R}$.
3. If $T_1 = k?\langle U_1\rangle;T_1'$, then $\mathsf{unfold}^n(T_2) = k?\langle U_2\rangle;T_2'$, $(T_1', T_2') \in \mathfrak{R}$ and $(U_2, U_1) \in \mathfrak{R}$.
4. If $T_1 = \mu \mathbf{t}.T$, then $(\mathsf{unfold}^1(T_1), T_2) \in \mathfrak{R}$.

where $\mathsf{unfold}^n(T)$ is the result of inductively unfolding the top level recursion up to a fixed level of nesting. The coinductive subtyping relation $T_1 <: T_2$ (read: T_1 is an *asynchronous subtype* of T_2) is defined when there exists a type simulation $\mathfrak{R}$ with $(T_1, T_2) \in \mathfrak{R}$. An output of T_1 can be simulated after applying asynchronous optimisation $\gg$ to the unfolded T_2. We also need to ensure object type U_1 is a subtype of U_2. This subtyping relation $T <: T'$ is decidable if all channels under each recursive prefix are distinct. $T^\star$ and T_{kernel} satisfy this condition since $r_1, s_1, t_1, u_1, r_2, s_2, t_2$ and u_2 are distinct under the recursive prefix.

To show that $T^\star <: T_{\text{kernel}}$, we start by unfolding T_{kernel} once to obtain

$$T_0 = r_1!\langle\rangle;s_1?\langle U\rangle;t_1?\langle\rangle;u_1!\langle U\rangle;r_2!\langle\rangle;s_2?\langle U\rangle;t_2?\langle\rangle;u_2!\langle U\rangle;T_{\text{kernel}}$$

Then $r_1!\langle\rangle$ matches the initial part of $T^\star$. To simulate the $r_2!\langle\rangle$ part of $T^\star$, $r_2!\langle\rangle$ is permuted by applying the asynchronous subtyping rules above, together with transitivity.

$$T_0 \gg T_0' = r_1!\langle\rangle;r_2!\langle\rangle;s_1?\langle U\rangle;t_1?\langle\rangle;u_1!\langle U\rangle;s_2?\langle U\rangle;t_2?\langle\rangle;u_2!\langle U\rangle;T_{\text{kernel}}.$$

Let $T^\star = r_1!\langle\rangle;r_2!\langle\rangle;T_R^\star$. Thus the unfold of $T_R^\star$ must be simulated by T'.

$$T' = s_1?\langle U\rangle;t_1?\langle\rangle;u_1!\langle U\rangle;s_2?\langle U\rangle;t_2?\langle\rangle;u_2!\langle U\rangle;T_{\text{kernel}}.$$

Next we unfold $T_R^\star$ as:

$$s_1?\langle U\rangle;t_1?\langle\rangle;u_1!\langle U\rangle;r_1!\langle\rangle;s_2?\langle U\rangle;t_2?\langle\rangle;u_2!\langle U\rangle;r_2!\langle\rangle;T_R^\star$$

The first three types $s_1?\langle U\rangle;t_1?\langle\rangle;u_1!\langle U\rangle$ can be simulated by T' in this order. However to simulate $r_1!\langle\rangle$ in above $T_R^\star$, T_{kernel} must be unfolded again since the type in front of T' does *not* include $r_1!\langle\rangle$ outside the recursive prefix. Hence we apply the asynchronous subtyping rule to solve the following relation:

$$\begin{aligned} r_1!\langle\rangle;s_2?\langle U\rangle;t_2?\langle\rangle;u_2!\langle U\rangle;r_2!\langle\rangle;T_R^\star <: {} & s_2?\langle U\rangle;t_2?\langle\rangle;u_2!\langle U\rangle; \\ & r_1!\langle\rangle;s_1?\langle U\rangle;t_1?\langle\rangle;u_1!\langle U\rangle; \\ & r_2!\langle\rangle;s_2?\langle U\rangle;t_2?\langle\rangle;u_2!\langle U\rangle;T_{\text{kernel}} \end{aligned}$$

By applying $\gg$ to the r.h.s., $r_1!\langle\rangle$ can be permuted to the top. Then we can use the input and output subtyping simulation rules in order to achieve the original pair $(T_R^\star,\ T')$ again. This concludes the verification of the double-buffering example.

Subject reduction for L1 is proved as in [33], just by replacing the standard branching subtyping relation in [20] to the one which incorporates asynchronous commutative subtyping in [33]. We can also obtain the three key correctness properties, communication safety, type safety and progress, as stated in [20, §5]. Hence we can formally show that the double-buffering example in L1 is correct with respect to these properties — neither deadlock, type-error nor communication mismatch can happen in the interactions among the three participants.

3.3 Further Safety Analysis

One of the key merits of the use of type signatures for interactions, multiparty session types, in the present compilation framework is that they enable and facilitate various safety analyses pertaining to communication actions (hence their DMA translations). One of such analyses is the following race freedom analysis, where we guarantee that, when communication operations in L1 are compiled into DMA primitives, no local writes will interfere with remote writes. This analysis is done at the L1 level. The net consequence is that, as far as the compiled code from L2 to L1 is statically checked to be safe by this analysis, its further compilation into L0 is ensured to be race-free.

We illustrate the basic idea via an example. Assume given three participants, (say) Alice, Bob and Carol, where Alice sends a boolean value to Bob, Bob sends an integer to Carol, and Carol sends another integer to Alice. Note there is a causal chain from the initial output by Alice to the final input, again at Alice.[6] Now assume the following is the program for Alice, with s_b its initial output channel to Bob and s_a its final input channel from Carol:

$$s_b!\langle \texttt{true} \rangle;\ \ldots;\ s_a?\langle x \rangle;\ \texttt{print}(x); \tag{3.1}$$

Now let us fill "…" in (3.1) as follows.

$$s_b!\langle \texttt{true} \rangle;\ \boxed{x := 5;}\ s_a?\langle x \rangle;\ \texttt{print}(x); \tag{3.2}$$

Assuming x is private, the coloured command can have a local write at x in parallel with the remote write at the same variable x, the latter represented as communication through the channel s_a but which is in effect carried out, in the compiled code, as a DMA write on x. This asynchronous remote write at x can take place concurrently as the local write at x by the command $x := 5$. Thus we do not know whether 5 or a different number by the remote write will get printed in the final print command.

Next we consider the following variation of the program above:

$$\boxed{x := 5;}\ s_b!\langle \texttt{true} \rangle;\ \boxed{y := 5;}\ s_a?\langle x \rangle;\ \texttt{print}(x); \tag{3.3}$$

In this case, assuming the causally chained communications among Alice, Bob and Carol as specified above, we have no racing at x (as far as x and y are not aliased). This is because we know Carol will write only after this program does the first output above, via s_b: as far as x is used for reading or writing *after* this prompting output via s_b — which will eventually initiate the asynchronous write at x via s_a — there can be no interference. Let us summarise this principle:

> *If a participant's output action is the cause of its subsequent input for a variable x, then using x* between *the prompting output and the subsequent input is dangerous. We want to prevent such dangerous occurrences of variables.*

Several observations are due:

[6] Such causal chains can be altered by permutations discussed in the previous subsection. Thus these chains need be extracted from the minimal local types of programs (which coincide with the principal types algorithmically inferred from untyped processes [33]).

– The safety property crucially depends on causality information (i.e. the relationship between an input and its prompting output) derived from session types.
– Once given this causality information, the standard control flow analysis can quickly check the existence/lack of such a dangerous path (modulo e.g. dead branches).
– This analysis can be done regardless of high-level languages in L2: it can be uniformly performed on all typable programs in L1.

This analysis is crucial for ensuring safety in the use of DMA operations. Note the analysis does not have to be performed for each L1 program: it suffices to ensure, once and for all, that a compilation from a given high-level language at L2 never induce dangerous processes in L1 in the above sense.

Another significant program analysis which can exploit the session type structures in L1 is the guarantee of the progress property, or of the lack of a deadlocked input. This property is immediately ensured when no two sessions interleave with each other, or no other blocking operations are present, which may often be the case in the compiled code. When two or more sessions can interleave, we can use many type-based and other analyses which can ensure the lack of deadlocks in communication, exploiting session type structures including its linearity.

There are other useful analyses depending on execution environments and kinds of applications, which will be discussed elsewhere.

4 Compiling Typed Processes to Distributed CMP

4.1 Basic Ideas

Processes with session types are guaranteed to follow rigorous communication structures, given as types. By tracing a session type, we know beforehand what and when processes will send and receive messages: we can even statically determine the target remote addresses of these communications. Such addresses can be exchanged at the time of session initiation.

Using this information, we can replace each message passing in a typed process with a direct remote write to the address of a variable in a core's local memory in a distributed memory CMP chip. As noted, the addresses of many of these variables can be known statically, hence can be exchanged at the time of session initiation. This allows an efficient execution of a conversation code, especially when a loop (iteration) is included inside a session. When one does need to treat dynamically generated data structures such as trees and graphs, whose size may not be able to be determined statically, one may also need to have dynamically allocated addresses communicated at runtime, for their use in subsequent communications. Note such addresses can be piggybacked in preceding messages in the same session.

Since our purpose is to have type-safe compilation, we use a (prototypical) typed low-level programming language targeted at distributed memory CMP and NoC [2, 10], which we call L0 for brevity. L0 is based on the C programming language, and features, among others:

– A two-level code structure where the outer level (called a section) encompasses all the code to run at a (virtual) core, and the inner level conventional C functions and variable/data declarations;

– a new type, **place**, denoting a core (that can be virtualised and mapped into available physical cores); and
– primitives to obtain an idle place (**newPlace**), to launch a new thread at some place (**spawn**), to obtain the current place (**here**), to asynchronously copy an array into some other place via DMA (**put**), and to wait for the completion of an incoming DMA operation (**wait**).

These constructs, together with the safety conditions for L1 programs, allow a direct translation from L1 programs to L0 programs. By the type-based analyses on L1 discussed in the preceding section, the resulting compiled code is guaranteed to satisfy key correctness properties such as synchronisation/type safety, race freedom, and progress, as far as we assume a correct compilation. Also note that the type annotations on the DMA operations in L0 coming from those in the original L1 programs enable us to perform type-based analyses on L0 programs independently.

Type information for multiparty sessions can be used not only at compile time from L1 to L0, but also at runtime. For example, process migration will become necessary from various needs for reconfiguration including load balancing. For this purpose, sound treatment of pending messages are essential, which can be assisted by precise information on the type signatures of involved conversations. In the following, we focus on the most basic usage of session type information in compilation to L0, i.e. compilation of session communications to safe and efficient DMA operations. Other usage of type information will be reported elsewhere.

4.2 Compilation

Figures 3 and 4 present a compilation of our running example into L0. As we have already observed, all typed message passing is replaced by DMA primitives, using addresses of the variables in the local memory of a target place for remote asynchronous write operations, where the addresses are shared by the *session initiation protocol* adapted for distributed memory CMP, as described below.

Section Main defines a program comprising a single procedure, necessarily named main. The program is uploaded at some (virtual) core and the execution of the main function starts. The first **spawn** instruction in Main.main copies section Kernel into the (virtual) core obtained previously via a call to the **newPlace** primitive (we assume this operation will block if no core is available), and launches the execution of Kernel.main.

The *session initiation* protocol works as follows: Kernel writes in variable a0 (received from Main at spawn time) a data-structure with two fields to be filled by the producer and the consumer. These fields are then passed to the respective places at **spawn** time. At this point both the producer and the consumer know the remote address of a variable in the kernel. They can now write in these variables the addresses of the data structures to be shared later, so that these components can communicate by writing to these addresses.

Section Producer comprises a local buffer to hold the produced data, two variables of type Sync (syncA, and syncB) used as a notification for safe DMA operation, and the variables for the target of the remote addresses of the communication. After the session initiation, the place running this section continuously fills the local buffer and

Fig. 3 L0 code for the double buffering example (Main and Producer sections)

```
typedef int[4096] Buffer;  // 16KBytes buffer
typedef struct {} Sync;
typedef struct {Buffer *bufferA, Buffer *bufferB} Buffers;
typedef struct {Sync *syncA, Sync *syncB} Syncs;

typedef struct {Syncs *syncs, Buffers buffers} ConsumerInit;
typedef struct {Buffers *buffers, Syncs syncs} ProducerInit;
typedef struct {ProducerInit *prod, ConsumerInit *cons} SessionInit;

section Main () {
  void main () {
    place mainPlace = here();
    place producer = newPlace();
    place consumer = newPlace();
    SessionInit a0;
    spawn Kernel(&a0, mainPlace, producer, consumer) at mainPlace;
    wait(&a0); // session initiation
    spawn Producer(a0.prod, mainPlace) at producer;
    spawn Consumer(a0.cons, mainPlace) at consumer;
}}

section Producer (ProducerInit *a1, place kernel) {
  Buffer buffer;
  Sync syncA; Sync syncB;
  Buffers kernelBuffers;

  void main () {
    put({&kernelBuffers, {&syncA, &syncB}}, a1, kernel); // session initiation
    wait(&kernelBuffers); // end session initiation
    produce: {
      // Produce buffer A
      foreach (i: 0..4095) buffer[i] = get_int();
      wait(&syncA);
      put(buffer, kernelBuffers.bufferA, kernel);
      // Produce buffer B
      foreach (i: 0..4095) buffer[i] = get_int();
      wait(&syncB);
      put(buffer, kernelBuffers.bufferB, kernel);
      loop produce;
    }}
}
```

puts it in one of the kernel's target buffers with a **put** instruction. A clearance, e.g., **wait**(&syncA), stating that the target buffer is ready must precede the actual placing of the data in the kernel's memory.

The Kernel section declares two incoming/outgoing buffers. After session initiation, it signals the producer that its buffers can now be written (the two instructions that precede the loop). It then waits for the completion of the DMA operation regarding the first of its buffers (bufferA), fills it with the XOR of each data element with the defined key and proceeds to write in the consumer memory, following the same wait-put protocol used by the producer before writing on the kernel's memory. Once the operation is completed, the kernel signals the producer that bufferA is ready to be re-written, proceeding to process bufferB.

Consumer should be easy to understand, it simply waits for the arrival of each buffer at the time, printing their contents.

Fig. 4 L0 code for the double buffering example (Kernel and Consumer sections)

```
section Kernel (SessionInit *a0, place mainPlace, place producer, place consumer){
  ProducerInit a1;
  ConsumerInit a2;
  Buffer bufferA; Sync syncA;
  Buffer bufferB; Sync syncB;
  int key = KEY;

  void main () {
    put({&a1, &a2}, a0, mainPlace);  // session initiation
    wait(&a1);
    wait(&a2);
    put({&bufferA, &bufferB}, a1.buffers, producer);
    put({&syncA, &syncB}, a2.syncs, consumer); // end session initiation
    put({}, a1.syncs.syncA, producer);
    put({}, a1.syncs.syncB, producer);
    process: {
      // Process buffer A
      wait(&bufferA);
      foreach (i: 0..4095) bufferA[i] = bufferA[i] ^ key;
      wait(&syncA);
      put(bufferA, a2.buffers.bufferA, consumer);
      put({}, a1.syncs.syncA, producer);
      // Process buffer B
      wait(&bufferB);
      foreach (i: 0..4095) bufferB[i] = bufferB[i] ^ key;
      wait(&syncB);
      put(bufferB, a2.buffers.bufferB, consumer);
      put({}, a1.syncs.syncB, producer);
      loop process;
   }}
}

section Consumer (ConsumerInit *a2, place kernel) {
  Buffer buffer;
  Syncs syncs;

  void main () {
    put({&syncs, {&buffer, &buffer}}, a2, kernel); // session initiation
    wait(&sync); // end session initiation
    consume: {
      // Consume buffer A
      put({}, syncs.syncA, kernel);
      wait(&buffer);
      printf("\nBuffer:\n");
      foreach (i: 0..4095) printf("%d ", buffer[i]);
      // Consume buffer B
      put({}, syncs.syncB, kernel);
      wait(&buffer);
      printf("\nBuffer:\n");
      foreach (i: 0..4095) printf("%d ", buffer[i]);
      loop consume;
   }}
}
```

The resulting code is in direct correspondence with the original typed processes in its operational structure, and, thanks to the well-typedness of the original process in L1 with respect to the declared session types, together with the static analysis for race freedom outlined in §3.3, we can show that the DMA operations in the resulting L0 code faithfully captures all and only communication and other behaviours as found in

the original L1 program, modulo the translation of the original session initiation into a protocol realising the equivalent functionality (which distribute remote addresses used for performing DMA writes: note these addresses in effect act as channel ends in the original process representation). In fact, the type-directed translation from L1 to L0 can annotate the resulting L0 code with types which closely correspond to those in the original L1 program. This type annotations make the resulting L0 code amenable to the type-based analyses isomorphic to those for L1 programs. This ensures, for the resulting L0 code, the aforementioned three key correctness properties, the synchronisation and race-error freedom, type-error freedom and progress.

We have developed a prototype compiler targeting for a IBM Cell Broadband Engine processor [14], so that we can compile high-level code to low-level code as in Figures 3 and 4, which can further be compiled and executed on Cell. More discussions on this implementation are given in Section 5.

4.3 Further Features

There are several key features of our intermediate language which we do not discuss in the present paper. In particular, although the example under consideration does not use shared session initialisation channels, we often need a component which accepts possibly concurrent requests for session initialisation at a shared channel from multiple clients. Such a channel may be located at main memory or at local memory of a distributed core. The shared server receives a request, at which point (for example) it may fork a thread to one of the available cores for serving the client's needs. Such a framework is especially important for realising shared services used by an unknown number of client processes, either inside an application or across applications, and demands an efficient treatment of possibly concurrent requests arriving at a same channel.

We can treat the arrival of such an indeterminate number of requests through several methods. As a simple way, each core may run a supervisor-mode process to which each user-level process may ask for communication to a shared channel in a remote core (note such requests tend to be relatively fewer than communications inside a session, so that a slightly higher cost for a shared request may be justified). Then a supervisor can put the request to its own queue in a remote or shared memory, which can be polled by a receiver of these requests. Putting a request in a queue can be followed by a simple notification. Such a scheme may be combined with mutual exclusion primitives (lock and/or compare and swap, see [45]) by multiple threads at the service process.

5 Conclusion

Conclusion and Further Topics. The translation from the initial simple stream application to the low-level code based on double buffering, through intermediate representation as typed processes, suggests flexibility in compilation and execution of concurrent programs in distributed CMP and other extremely concurrent computing environments, opening new opportunities and challenges. We already mentioned the use of our recent work [33] in our compilation framework, which is based on a subtyping relation on multiparty session types which are generalised to capture asynchrony as found in

the double buffering process above. Further development of the compilation framework will necessitate new compilation and static analysis techniques for inherently concurrent code, a new, scalable runtime framework for dynamic allocations of hardware resources to communicating processes making the best of their type structures, a formal guarantee of correctness properties for such a runtime, an effective threads scheduling mechanism in each local core, protection and security mechanisms, and integration and management of different abstractions for concurrency.

Related Work. There are several recent works which are closely related and will complement the approaches taken in the presented research direction: research from multiple directions will be needed to explore the rich field of structured concurrent programming. Among these related works, we list only a few. Occam-Pi [46] offers a highly efficient language architecture for channel-based concurrency with potentially millions of light-weight processes. Sing#, a derivative of C# developed for Singularity OS [11], uses a variant of session types called *contracts* to specify the interfaces between OS components, which communicate via channel-based message passing in shared memory environments. X10 [6] presents an advanced language constructs for structured, typed concurrent imperative programming for partitioned shared memory with high-performance computing as its application domain. Kilim [40] is an actor framework for Java based on cooperatively-scheduled lightweight threads which communicate by message-passing. StreamFlex [39] is a real-time stream API for Java guaranteeing sub-millisecond response times and type safety, using a type-based classification of heap objects to obtain a high throughput. In all these languages, high-level structuring constructs play an essential role not only for clean description of concurrency but also for efficient program execution.

A preliminary version of this paper was presented in [18].

Implementation Status. We are currently working on the experiments of the general framework proposed in the present paper. It centres on a simple imperative concurrent language equipped with multiparty session communications and their types, which is close to the language we discussed in Section 3. The language, combined with two other associated languages, is intended to serve as an intermediate language (roughly of level L1 in Section 1), to which typed high-level concurrent languages such as X10 [6], StreamIt [42] and others are compiled into.

The current framework implements a series of type-directed translation steps from high-level typed concurrent languages into C-code targeted at the Cell Broadband Engine architecture. Our experiments so far have been restricted to a single Cell processor. Current efforts focus on, among others, providing support for the deploying of applications across processors on the same blade and across blades. For that purpose we are using a cluster of three IBM QS21 bladecenters [22] and their compiler architecture.

Acknowledgements. The work is partially supported by the EU IST proactive initiative FET-Global Computing (projects Sensoria and Mobius), as well as the Treaty of Windsor Anglo-Portuguese Joint Research Programme B–4/08. The first and the last authors are partially supported by EPSRC GR/T03208, GR/T03215, EP/F002114 and

EP/F003757. They thank Francisco Martins for advice and suggestions on the intermediate languages and compilation scheme.

References

1. Bowen Alpern, Larry Carter, and Jeanne Ferrante. Modeling parallel computers as memory hierarchies. In *Proceedings of Programming Models for Massively Parallel Computers*, pages 116–123. IEEE Computer Society Press, 1993.
2. Luca Benini and Giovanni De Micheli. Networks on chip: a new SoC paradigm. *IEEE Computer*, 35:1, 2002.
3. Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global Progress in Dynamically Interleaved Multiparty Sessions. In *CONCUR'08*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
4. G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, 1996.
5. Eduardo Bonelli and Adriana Compagnoni. Multipoint Session Types for a Distributed Calculus. In *TGC'07*, volume 4912 of *LNCS*, pages 240–256, 2008.
6. Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05*, pages 519–538. ACM Press, 2005.
7. David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, 1993.
8. David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., 1997.
9. William J. Dally. Enabling technology for on-chip interconnection networks. In *NOCS '07*, page 3. IEEE Computer Society, 2007.
10. William J. Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. In *DAC '01*, pages 684–689. IEEE Computer Society, 2001.
11. Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity OS. In *EuroSys '06*, pages 177–190. ACM Press, 2006.
12. Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the Memory Hierarchy. In *SC'06*, page 83. ACM Press, 2006.
13. Pat Gelsinger, Paolo Gargini, Gerhard Parker, and Albert Yu. Microprocessors circa 2000. *IEEE SPectrum*, pages 43–47, 1989.
14. Michael Gschwind. The Cell Broadband Engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, 35(3):233–262, 2007.
15. Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.
16. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, 1983.
17. Tony Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
18. Kohei Honda, Vasco Vasconcelos, and Nobuko Yoshida. Type-directed compilation for multicore programming. In *PLACES '08*, ENTCS. Elsevier, 2009. To appear.
19. Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

20. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.
21. IBM. ALF double buffering. `http://www.ibm.com/developerworks/blogs/page/powerarchitecture?entry=ib%omb_alf_sdk30_5`.
22. IBM. IBM BladeCenter QS21. `http://www-03.ibm.com/systems/bladecenter/hardware/servers/qs21/index.h%tml`.
23. IBM. Cell broadband engine programming tutorial version 2.0, 2006.
24. Ujval J. Kapasi, William J. Dally, Scott Rixner, John D. Owens, and Brucek Khailany. The imagine stream processor. In *ICCD '02*, pages 282–288, 2002.
25. Chetana N. Keltcher, Kevin J. McGrath, Ardsher Ahmed, and Pat Conway. The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66–76, 2003.
26. Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, 2006.
27. Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, 2005.
28. Chuan-Kai Lin and Andrew P. Black. DirectFlow: A domain-specific language for information-flow systems. In *ECOOP '07*, volume 4609 of *LNCS*, pages 299–322. Springer, 2007.
29. David B. Loveman. Program improvement by source to source transformation. In *POPL '76*, pages 140–152. ACM Press, 1976.
30. Robin Milner. Processes, a mathematical model of computing agents. In *Logic Colloquium, Bristol 1973*, pages 157–174. North Holland, Amsterdam, 1975.
31. Robin Milner. Functions as processes. In *ICALP'90*, volume 443 of *LNCS*, pages 167–180. Springer, 1990.
32. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.
33. Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global Principal Typing in Partially Commutative Asynchronous Sessions. In *ESOP'09*, volume 1782 of *LNCS*. Springer, 2009.
34. Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *ASPLOS-VII*, pages 2–11. ACM Press, 1996.
35. D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor. In *ISSCC '05*, volume 1, pages 184–592, 2005.
36. Fred J. Pollack. New microarchitecture challenges in the coming generations of CMOS process technologies. In *MICRO '99*, page 2. IEEE Computer Society, 1999.
37. José Carlos Sancho and Darren J. Kerbyson. Analysis of Double Buffering on two Different Multicore Architectures: Quad-core Opteron and the Cell-BE. In *IPDPS '08*. IEEE, 2008.
38. Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 1993.
39. Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. StreamFlex: High-Throughput Stream Programming in Java. In *OOPSLA '07*, pages 211–228. ACM Press, 2007.
40. Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for Java. In *ECOOP '08*, volume 5142 of *LNCS*, pages 104–128. Springer, 2008.
41. Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.
42. William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *CC '02*, volume 2304 of *LNCS*, pages 179–196. Springer, 2002.

43. William Thies, Michal Karczmarek, Janis Sermulins, Rodric Rabbah, and Saman Amarasinghe. Teleport messaging for distributed stream programs. In *PPoPP'05*, pages 224–235. ACM Press, 2005.
44. S.R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1):29–41, 2008.
45. Vasco T. Vasconcelos and Francisco Martins. A multithreaded typed assembly language. In *Proceedings of TV'06 - Multithreading in Hardware and Software: Formal Approaches to Design and Verification*, 2006.
46. Peter Welch and Fred Barnes. Communicating Mobile Processes: introducing occam-pi. In *25 Years of CSP*, volume 3525 of *LNCS*, pages 175–210. Springer, 2005.