Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 241 (2009) 101–111
www.elsevier.com/locate/entcs

Type-Directed Compilation for Multicore Programming

Kohei Honda[1]

Queen Mary, University of London

Vasco T. Vasconcelos[2]

University of Lisbon

Nobuko Yoshida[3]

Imperial College London

Abstract

In this paper, we outline a general picture of our ongoing work on a compilation and execution framework for a class of multicore CPUs [10,21,22]. Our focus is to harness the power of concurrency and asynchrony in one of the major forms of multicore CPUs based on distributed, noncoherent memory [22], using the well-known technology of type-directed compilation [19]. 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 [12,26], we obtain both type-safe and efficient compilation into processes distributed over multiple cores with local memories.

1 Backgrounds

1.1 Concurrency at the Cores of Computing

In spite of the applications' increasing reliance on distributed components in the Internet and the world-wide web, the basic computing paradigm in software development has been centring on monolithic, predominantly sequential code. This fits our hardware, which is a virtually monolithic Von Neumann Machine (VNM), even though interactions with the distributed services often necessitate the use of concurrent threads inside a program.

[1] Email: kohei@dcs.qmul.ac.uk
[2] Email: vv@di.fc.ul.pt
[3] Email: yoshida@doc.ic.ac.uk

doi:10.1016/j.entcs.2009.06.006

It is only during the last decade that limiting physical parameters in VLSI manufacturing process [10,21,23] started to push a fundamental change in the internal environment of computing machinery, from monolithic Von Neumann architectures to concurrent ones, the so-called chip-level multiprocessing (CMP from now on), giving rise to CPUs with multiple cores. A multicore CPU is most effectively utilised by having multiple modules running concurrently, even inside a single application. Combined with the increasing reliance on distributed components through web services and sensor networks, computing is now becoming concurrent inside out.

1.2 A Machine Model for CMP

Following the standard dichotomy in parallel computer architecture [6], a multicore CPU can be categorised in the spectrum ranging from SMP-like coherent cache architecture, cf. [16,28], to distributed, non-coherent memory, cf. [22]. In the former, memory coherence is maintained across multiple cores, while in the latter, sharing of data among cores is performed by explicit instructions on non-uniform memory space. This second form is often found in multiprocessor system-on-chips (MPSoCs) for embedded systems, one of the areas where multicore CPUs are being effectively deployed centring on a flexible on-chip interconnect.

A non-uniform cache access can be realised by different methods such as cache-line locking. One basic method employs direct asynchronous data transfer, or Direct Memory Access (DMA), to an on-chip memory local to each core. A central observation underlying this approach is that trying to annihilate distance (i.e. to maintain strict coherence) is too costly, reminiscent of the observations that coherent distributed shared memory over a large number of nodes in a cluster is hard to provide as a hardware-level interface to programmers. Thus we regard CMP as distributed VNMs, without built-in memory coherence mechanisms, along the lines of the LogP model [5] and PGAS [4].

Because of its efficiency and flexibility, this framework is widely used in MPSoC for embedded systems, including a major multicore chip [22]. It is a natural model when we consider CMP as a microscopic form of distributed computing, suggesting its potential scalability when the number of cores per chip increases. Further it can realise arbitrary forms of data sharing among cores, and in that sense it is general-purpose. Being efficient and general-purpose, however, this computing model is also known to be extremely hard and unsafe to program. Indeed, the very element that makes the major mode of data sharing in this model, DMA, fast and general-purpose, also makes it unwieldy and dangerous: it involves raw writes of one memory area to another, asynchronously issued and asynchronously performed. Communications programming is hard even with high-level, type-safe languages, leading to bugs such as synchronisation errors and deadlocks. Communications on distributed CMP are untyped operations which directly (and asynchronously) write byte sequences to regions of cores' memories, so that errors in communications easily destroy the work being conducted in multiple cores. It is thus paramount for exploiting the power of distributed CMP that we can guarantee safe data transfer (DMA communication) without compromising its efficiency.

While there are other challenges in programming distributed memory CMP, we identify this issue as a central one, whose solution may offer a basis for solving other issues. In the following sections we shall argue for the use of types for interaction for addressing this issue. Throughout the discussions we consider an idealised model along the lines of [5], where a chip consists of multiple isomorphic VNMs, of the same ISA and each with its own memory. Data sharing is through a basic form of DMA, asynchronous copy of possibly multiple contiguous words from one memory to another. For simplicity we do not take into consideration either the size of local memory or the maximum unit of transfer [5], and discuss only a so-called "push" version of DMA (cf. [22]).

2 From Applications to Typed Communicating Processes

2.1 A Type-Directed Compilation Framework

One of the key features of CMP in general, including cache-coherent and distributed memory variants, is its versatility to host a variety of applications, in size, in granularity of parallelism, and in the shape of control and data flows. Such applications may be written using domain specific languages [17,25]. How can we translate these high-level applications to executables for CMP? The basic idea of our approach is to stipulate *typed communicating processes* as a language for an intermediate compilation step, and perform a *type-directed compilation* [19] onto a typed machine language for CMP. Schematically:

$$\begin{array}{c}\text{high-level concurrent languages } (\mathsf{L2}) \\ \Downarrow \\ \text{typed imperative processes } (\mathsf{L1}) \\ \Downarrow \\ \text{typed CMP executable } (\mathsf{L0})\end{array}$$

Above L0, L1, L2 refer to abstraction levels. Each $\Downarrow$ stands for one or more type-preserving compilations. A central idea of the proposed framework is the use of, at L1, an intermediate concurrent imperative language with types for channel-based conversations, combined with typed shared memory primitives. For communication types, it uses a variant of session types [12,26] for multiparty interactions [2,3,13], into which existing high-level session types can be easily translated, and which allows efficient and safety-preserving compilation to distributed CMP primitives. Types at L1 will be generated from types in L2 together with interaction structures arising in the translation process, as we shall illustrate with a concrete example below.

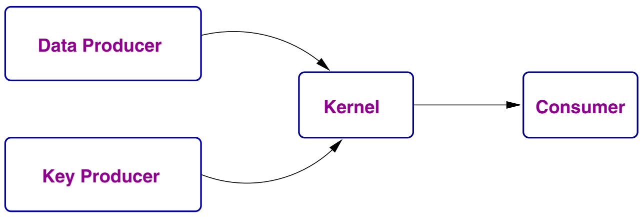

Fig. 1. A simple program for stream cipher

2.2 Streaming Example

We take the simple program for stream cipher [24], depicted in Figure 1. Data Producer and KeyProducer continuously send a data stream and a key stream respectively to Kernel. Kernel calculates their XOR and sends the result to Consumer. A high-level specification of such an example — specifying kernels and their connections through asynchronous streams as Kahn's networks — can be written using a DSL for streaming [17,25], which we omit. Our purpose is to translate this program to a type-safe low-level program for distributed memory CMP.

2.3 Processes with Session Types

We use imperative processes with session types [12,26] as an intermediate language. Our motivations are two-fold. First it offers an effective *source* language for compilation into a typed assembly language for CMP, as we shall outline soon. Secondly 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.

Below we show a process representation of the streaming algorithm given above. In order to illustrate the key ideas, we use a simple translation scheme. In practice we use a slightly more complex, and more efficient, translation, which we shall briefly discuss at the end. The kernel initiates a session:

$$\begin{aligned}\mathrm{Kernel} \overset{\mathrm{def}}{=} \ & \mathsf{def}\ \mathrm{K}(d,k,c) \ = \ d!\langle\rangle;\ k!\langle\rangle;\ d?(x);\ k?(y);\ c?();\ c!\langle x\ \mathsf{xor}\ y\rangle;\ \mathrm{K}\langle d,k,c\rangle \\ & \mathsf{in}\ \ \overline{a}(d,k,c).\mathrm{K}\langle d,k,c\rangle\end{aligned}$$

Observe that the channels d and k are used for Kernel to receive data and keys from Data Producer and Key Producer, respectively. Before receiving, Kernel notifies Data/Key Producers that it is ready before receiving data/keys. Such an insertion of a notification message before the reception of datum is essential for safe translation into DMA operations, since without them, a datum may be written to a memory region while that region is being read and/or written by a local process, leading to inconsistent values. Note also these signals are necessary even when we use the class of streaming languages with the most regular behaviour such as static and cyclo-static data flow languages.

During the compilation of stream programs, we also perform standard strip mining [18], transforming streams into strips of large chunks of data (above, x and y may as well contain large arrays, say 16KB).

The channel c is used for Consumer to receive the encrypted data from Kernel, which is also used for notifying its readiness to receive the data. The keyword def denotes a recursive process; $\overline{a}(d,k,c)$ is a session initiation establishing a session between the three parties; $d?(x)$ is an input action at d; and $c!\langle x \ \mathsf{xor}\ y\rangle$ is an output action at c.

DataProducer and Consumer can be given as follows.

$$\text{DataProducer} \overset{\text{def}}{=} \mathsf{def}\ \mathrm{P}(d,k,c) = d?();d!\langle data\rangle \mathrm{P}\langle d,k,c\rangle\ \mathsf{in}\ a(d,k,c).\mathrm{P}\langle d,k,c\rangle$$

$$\text{Consumer} \overset{\text{def}}{=} \mathsf{def}\ \mathrm{C}(d,k,c) = c!\langle\rangle\ c?(data);\mathrm{C}\langle d,k,c\rangle\ \ \mathsf{in}\ \ a(d,k,c).\mathrm{C}\langle d,k,c\rangle$$

KeyProducer is identical to DataProducer except that it outputs at k rather than at d.

In all these processes, we assume that output actions of these processes are asynchronous (no blocking), and that input actions are synchronous. When these three processes are composed, messages are always consumed in the order they are produced because of the linearised usage of each channel.

The exchange of messages as above forms a “conversation” among processes, with a precise structure: this structure we abstract below as a type. The session type of the Kernel is given as:

$$T_K \ = \ \mu\mathbf{t}.d!\,\langle\rangle;k!\,\langle\rangle;d?\,\langle\mathsf{bool}\rangle;k?\,\langle\mathsf{bool}\rangle;c?\,\langle\rangle;c!\,\langle\mathsf{bool}\rangle;\mathbf{t}$$

Above $\mu\mathbf{t}.T$ represents a recursive type, $k?\,\langle\mathsf{bool}\rangle$ (resp. $k!\,\langle\mathsf{bool}\rangle$) denotes the input (resp. output) of a value of bool-type, and $T;T'$ denotes a sequencing. The type of the DataProducer is given as $\mu\mathbf{t}.d?\,\langle\rangle;d!\,\langle\mathsf{bool}\rangle;\mathbf{t}$. Similarly for KeyProducer and Consumer. Safe parallel composition of communicating code is guaranteed by checking duality of types: the type of the Kernel and one of the DataProducer are dual to each other at d, so that there is no communication error occurs at d. Similarly for k and c.

3 Compiling Typed Processes to Asynchronous CMP

3.1 Type-Directed Compilation

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 process will send and receive as messages, as well as the target remote addresses of these communications. Using this information, we can replace each message passing in a typed processes with a direct remote write to the address of a variable in a core’s local memory in a multicore chip. Since our purpose is to have type-safe compilation, we use a prototypical programming language targeted at distributed

```
typedef int[1024] Buffer;
typedef struct {} Sync;
typedef struct {Sync **sync, Buffer *buffer} ConsumerInit;
typedef struct {Buffer **buffer, Sync *sync} ProducerInit;
typedef struct {ProducerInit *data, ProducerInit *key, ConsumerInit *cons}
           KernelInit;

section Main () {
  void main () {
    place mainPlace = here();
    place dataProducer = newPlace();
    place keyProducer = newPlace();
    place consumer = newPlace();
    KernelInit a0;
    spawn Kernel(&a0,mainPlace,dataProducer,keyProducer,consumer) at mainPlace;
    wait(&a0); // session initiation
    spawn DataProducer(a0.data, mainPlace) at dataProducer;
    spawn KeyProducer(a0.key, mainPlace) at keyProducer;
    spawn Consumer(a0.cons, mainPlace) at consumer;
}}

section Kernel (KernelInit *a0, place mainPlace, place dataProducer,
    place keyProducer, place consumer) {
  ProducerInit a1, ProducerInit a2, ConsumerInit a3;
  Buffer keys, Buffer buffer, Sync sync;
  void main () {
    put({&a1, &a2, &a3}, a0, mainPlace);  // begin session initiation
    wait(&a1); wait(&a2); wait(&a3);
    put(&buffer, a1.buffer, dataProducer);
    put(&keys, a2.buffer, keyProducer);
    put(&sync, a3.sync, consumer); // end session initiation
    loop: {
      put({}, a1.sync, dataProducer);
      put({}, a2.sync, keyProducer);
      wait(&keys); wait(&buffer);
      foreach (i: 0..1023) buffer[i] = buffer[i] ^ keys[i];
      wait(&sync);
      put(buffer, a3.buffer, consumer);
      jump loop;
}}}

section DataProducer (ProducerInit *a1, place kernel) {
  Sync sync, Buffer buffer, Buffer *kernelBuffer;
  void main () {
    put({&kernelBuffer, &sync}, a1, kernel); // begin session initiation
    wait(&kernelBuffer); // end session initiation
    loop: {
      foreach (i: 0..1023) buffer[i] = get_byte()
      wait(&sync);
      put(buffer, kernelBuffer, kernel);
      jump loop;
}}}

section Consumer (ConsumerInit *a3, place kernel) {
  Buffer buffer, Sync* sync;
  void main () {
    put({&sync, &buffer}, a3, kernel); // begin session initiation
    wait(&sync); // end session initiation
    loop: {
      put({}, sync, kernel);
      wait(&buffer);
      printf("\nBuffer:\n");
      foreach (i: 0..1023) printf("%d ", buffer[i]);
      jump loop;
}}}
```

Fig. 2. L0 code for the streaming example (key producer is similar to data producer)

memory CMP and NoC [1,7], 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 core, and the inner level conventional C functions and variable/data declarations;
- a new type, **place**, denoting a core; and
- primitives to acquire (the address of a) core (newPlace), to launch a new thread at some core (**spawn**), to obtain the current core (here), to asynchronous copy an array into some other core via DMA (put), and to wait for the completion of an incoming DMA operation (wait).

Figure 2 presents one possible result of compiling our running example into L0. As we observed, all typed message passing is replaced by DMA primitives, using addresses of the variables in the local memory of a target core for remote asynchronous write operations, where the addresses are shared by a *session initiation protocol*, which describe below.

Section Main defines a program comprising a single procedure, necessarily named main. The program is intended to be uploaded at some core and the execution of the main function started. The first **spawn** instruction in Main.main copies section Kernel into the core obtained previously via a call to the newPlace() primitive (which we assume to block if no core, which may as well be virtualised, is available), and launches the execution of function Kernel.main.

The *session initiation* protocol works as follows: Kernel writes in variable a0 (received from Main at spawn time) a struct with space for three fields to be filled by the two producers and the consumer. These fields are then passed to the respective places at **spawn** time. At this point each of the producers/consumer knows the remote address of a variable in the kernel. These cores can now write in kernel variables the addresses of the data-structures that are to be shared later, so that these components can communicate by writing to these shared addresses (for which the lack of conflict, or racing, is guaranteed by session types).

The data producer section comprises a buffer to hold its data, an empty struct used as a notification message for safe DMA operation, and a single function (necessarily named main). After session initiation, the core running this program fills its buffer, waits for a clearance to proceed from the kernel, wait(&sync), and puts its buffer in the kernel's memory with a put instruction.

The kernel program declares two buffers (one incoming, another incoming/outgoing). After session initiation, the kernel signals the producers that its buffers can now be written (the two first put instructions in the loop), and waits for the completion of the DMA operation (the two next wait instructions). Then fills its buffer with the XOR of data and key and proceeds to write in the consumer memory, following the same wait-put protocol used by the producers before writing on the kernel's memory.

The consumer should be easy to understand; the code for the key producer, absent, is similar to the data producer. We have developed a prototype compiler

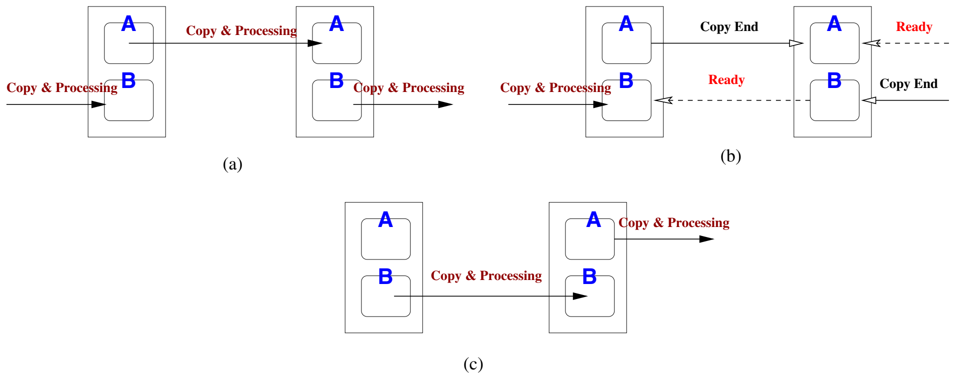

Fig. 3. Double-Buffering

for L0; the code in Figure 2 was run on a 3 blades of model QS21.

3.2 Shared Channels

The example under consideration does not use shared access to main memory. However, it is natural that a program which accepts multiple requests at a shared channel (located at main memory), receives a request, then forks a thread to one of the available cores. Generally this demands multiple clients to invoke a shared channel concurrently. In this and related schemes, a shared initial channel can be effectively realised by the combination of traditional load and store instructions together with mutual exclusion primitives (lock [29] or compare and swap) and DMAs.

4 Double Buffering Stream Processing

Our approach is based on a simple premise: session types offer rigorous abstraction of conversation structures, and, as far as concurrent programs can be represented as a collection of conversations, we can use their types in order to realise these conversations through asynchronous data transfers among local memories of multiple cores. Processes offer readable, transparent program structures, as well as a target of translation, and types guarantee type-safety of compiled code, including the lack of data corruption in spite of direct data transfers to memory regions of local cores.

There are several topics which we could not discuss in this paper. We however briefly touch one topic, which is important for practice. The process-based representation of streaming programs we presented in the previous section is inefficient due to the lack of asynchrony: the processing and transfer of data are done strictly in turn, so that processors waste time whenever transfer is taking place. Thus it fails to make the best of throughput inherent in distributed memory CMP. We thus transform the protocol structure slightly.

For brevity we consider three-party interactions, from a single source to the kernel to the consumer, and only present the session type of the kernel. Let s be a channel used for data-transfer with the right-hand side, while k is with the left-hand side; and "$s \triangleleft \mathsf{ReadyA}$;" (resp. "$s \triangleright \mathsf{ReadyA}$") sends (resp. receives) a signal which tells A is empty.

$$
\begin{array}{l}
s \lhd \mathsf{ReadyA}; s \lhd \mathsf{ReadyB};\\
\quad \mu\mathbf{t}.s?\,\langle T\rangle; k \rhd \mathsf{ReadyA}; k!\,\langle T\rangle; s \lhd \mathsf{ReadyA}; s?\,\langle T\rangle; k \rhd \mathsf{ReadyB}; k!\,\langle T\rangle; s \lhd \mathsf{ReadyB}; \mathbf{t}
\end{array}
$$

This type says: first it sends signal to s; then it gets the data into A from s; once the data transfer is completed *and* it gets the signal to tell A is free from k, then it starts transferring the data to k; similarly for B. But we do not wish to wait for A to receive the signal from the right-hand-side to start receiving the data into B from the left-hand-side via s. In fact, using the subtyping relation discussed in [20], the following type can match with the above type.

$$\mu\mathbf{t}.s \rhd \mathsf{ReadyA}; s \rhd \mathsf{ReadyB}; s!\,\langle T\rangle; s!\,\langle T\rangle; \mathbf{t}$$

This scheme is essentially the standard double buffering technique used in stream/-media processing and high-performance, multicore computing [14,15]. We depict its general protocol structure in Figure 3. Using the new session type, stream programs can now get compiled into their (type-safe) double-buffering implementations. Here processing can be continuously done while communications (data transfer by DMA) are being performed, while synchronisation signals are making these communications safe. Thus we obtain both the type and communication safety and efficiency comparable to the standard untyped double buffering code. Observe also that, while this new translation does change the communiction behaviour of each component, the whole application still preserves the same input/output behaviour.

5 Conclusion

The transformation of the initial simple translation into the one based on double buffering suggests flexibility in compilation and execution of concurrent programs in CMP and other extremely concurrent computing environments, opening new opportunities and challenges. We need more flexibility and generality in type structures for capturing varied protocols (for example our recent aforementioned work [20] discusses a subtyping relation on multiparty session types which are generalised to capture asynchrony as found in the double buffering process above), new compilation and static analysis techniques, new runtime architectures which can cater for, for example, dynamic allocations of hardware resources to processes, and new abstractions. Research from multiple directions (here we only refer to [4,8,9] among many closely related and/or complementary works) will be needed to explore the rich field of structured concurrent programming. We expect that basic type structures associated with language constructs will play a fundamental role in this problem domain, based on which other analysis and validation methods can be exploited.

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 2), to which typed high-level concurrent languages such as X10 [4], StreamIt [27] and others are compiled into. The details of this language and its typing system are discussed in the full version [11]. Among others, the framework implements a series of type-directed translation steps from high-level typed concurrent languages into C-code targeted at the Cell architecture, using IBM's QS21 blade servers and their compiler architecture.

Acknowledgement

We are grateful to a reviewer for useful comments, and to Hervé Paulino for contributing the code in Figure 2. Special thanks go to Francisco Martins, Dimitris Mostrous, Hervé Paulino and Tiago Cogumbreiro for the ongoing collaboration on the development of the compiler and execution framework for concurrent languages in a distributed memory CMP (the Cell architecture), exploring feasibility of the ideas proposed in this paper. The work is partially supported by British Council Treaty of Windsor Anglo-Portuguese Joint Research Programme, EPSRC GR/T03208, GR/T03215, EP/F002114, EP/F003757, IST2005–015905 MOBIUS, and IST2005–016004 Sensoria.

References

[1] Benini, L. and G. D. Micheli, *Networks on chip: a new SoC paradigm*, IEEE Computer **35:1** (2002).

[2] Bettini, L., M. Coppo, L. D'Antoni, M. De Luca, M. Dezani-Ciancaglini and N. Yoshida, *Global Progress in Dynamically Interleaved Multiparty Sessions*, in: *CONCUR'08*, LNCS **5201** (2008), pp. 418–433.

[3] Bonelli, E. and A. Compagnoni, *Multipoint Session Types for a Distributed Calculus*, in: G. Barthe and C. Fournet, editors, *TGC'07*, LNCS **4912** (2008), pp. 240–256.

[4] Charles, P., C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun and V. Sarkar, *X10: an object-oriented approach to non-uniform cluster computing*, in: *OOPSLA*, 2005, pp. 519–538.

[5] Culler, D., R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian and T. von Eicken, *Logp: towards a realistic model of parallel computation*, SIGPLAN Not. **28** (1993), pp. 1–12.

[6] Culler, D. E., A. Gupta and J. P. Singh, "Parallel Computer Architecture: A Hardware/Software Approach," Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[7] Dally, W. J. and B. Towles, *Route packets, not wires: On-chip interconnection networks*, in: *DAC*, 2001, pp. 684–689.

[8] Ennals, R., R. Sharp and A. Mycroft, *Linear types for packet processing*, in: *ESOP*, 2004, pp. 204–218.

[9] Fähndrich, M., M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi, *Language support for fast and reliable message-based communication in singularity os*, in: *EuroSys2006*, ACM SIGOPS (2006), pp. 177–190.

[10] Gelsinger, P., P. Gargini, G. Parker and A. Yu, *Microprocessors circa 2000*, IEEE SPectrum (1989), pp. 43–47.

[11] Honda, K., V. Vasconcelos and N. Yoshida, *Low-level session types for multicore programming*, full version, 150 pages.

[12] Honda, K., V. T. Vasconcelos and M. Kubo, *Language primitives and type disciplines for structured communication-based programming*, in: *ESOP'98*, LNCS **1381** (1998), pp. 22–138.

[13] Honda, K., N. Yoshida and M. Carbone, *Multiparty Asynchronous Session Types*, in: G. C. Necula and P. Wadler, editors, *POPL'08* (2008), pp. 273–284.

[14] IBM, *ALF double buffering*, `http://www.ibm.com/developerworks/blogs/page/powerarchitecture?entry=ibomb_alf_sdk30_5`.

[15] José Carlos Sancho and Darren J. Kerbyson, *Analysis of Double Buffering on two Different Multicore Architectures: Quad-core Opteron and the Cell-BE*, in: *IEEE/ACM Int. Parallel and Distributed Processing Symposium (IPDPS)* (2008), pp. 1–12.

[16] Kongetira, P., K. Aingaran and K. Olukotun, *Niagara: A 32-way multithreaded sparc processor*, IEEE Micro **25** (2005), pp. 21–29.

[17] Lin, C.-K. and A. P. Black, *DirectFlow: A domain-specific language for information-flow systems*, in: *ECOOP*, LNCS **4609** (2007), pp. 299–322.

[18] Loveman, D. B., *Program improvement by source to source transformation*, in: *POPL '76* (1976), pp. 140–152.

[19] Morrisett, G., D. Walker, K. Crary and N. Glew, *From System F to typed assembly language*, ACM Trans. Program. Lang. Syst. **21** (1999), pp. 527–568.

[20] Mostrous, D., N. Yoshida and K. Honda, *Global Principal Typing in Partially Commutative Asynchronous Sessions*, in: *ESOP'09*, LNCS (2009), to appear.

[21] Olukotun, K., B. A. Nayfeh, L. Hammond, K. G. Wilson and K. Chang, *The case for a single-chip multiprocessor*, in: *ASPLOS*, 1996, pp. 2–11.

[22] Pham, D. et al., *The design and implementation of a first-generation cell processor*, in: *ISSCC Dig. Tech. Papers* (2005), pp. 184–185.

[23] Pollack, F. J., *New microarchitecture challenges in the coming generations of cmos process technologies*, in: *MICRO* (1999), p. 2.

[24] Schneier, B., "Applied Cryptography: Protocols, Algorithms, and Source Code in C," John Wiley & Sons, Inc., 1993.

[25] Spring, J. H., J. Privat, R. Guerraoui and J. Vitek, *StreamFlex: high-throughput stream programming in Java*, in: *OOPSLA* (2007), pp. 211–228.

[26] Takeuchi, K., K. Honda and M. Kubo, *An interaction-based language and its typing system*, in: *PARLE'94*, LNCS **817** (1994), pp. 398–413.

[27] Thies, W., M. Karczmarek and S. P. Amarasinghe, *Streamit: A language for streaming applications*, in: *CC '02: Proceedings of the 11th International Conference on Compiler Construction* (2002), pp. 179–196.

[28] Vangal, S., 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*, Solid-State Circuits, IEEE Journal of **43** (Jan. 2008), pp. 29–41.

[29] Vasconcelos, V. T. and F. Martins, *A multithreaded typed assembly language*, in: *Proceedings of TV06 - Multithreading in Hardware and Software: Formal Approaches to Design and Verification*, 2006, pp. 133–141.