Type-Directed Compilation for Multicore Programming

Kohei Honda
Queen Mary Univ. of London

Vasco T. Vasconcelos
University of Lisbon

Nobuko Yoshida
Imperial College London

Preamble. In this abstract we outline a general picture of our ongoing work on compilation into multicore CPUs [8, 14, 15]. Our focus is to harness the power of concurrency and asynchrony in one of the major forms of multicore CPUs based on noncoherent shared memory, using the well-known technology of type-directed compilation [13]. The key idea is to regard explicit asynchronous data transfer among local caches as a realiser of communication among processes. By typing processes with a variant of session types [9, 19], we obtain both type-safe and efficient compilation into processes distributed over multiple cores.

Concurrency at the Cores of Computing. In spite of the increasing reliance on distributed components in the Internet and the world-wide web, the basic computing paradigm for our applications had 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.

It is only during the last decade that limiting physical parameters in VLSI manufacturing process [8, 14, 16] 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 programs 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.

A Machine Model for CMP. Following the standard dichotomy in parallel computer architecture [4], a multicore CPU can be based on either coherent cache (or SMP), cf. [11], or non-coherent cache (or non-cache-coherent NUMA), cf. [15]. In the former, memory coherence is maintained across multiple cores, while in the latter, sharing of data among cores is performed in 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, just as coherent distributed shared memory over a large number of nodes is unfeasible. Thus we regard CMP as distributed VNMs, along the lines of the LogP model [3] and PGAS [2].

Because of its efficiency and versatility, this framework is widely used in MPSoC for embedded systems, as noted already, including a major multicore chip [15]. 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, which can easily destroy the works being conducted in multiple cores.

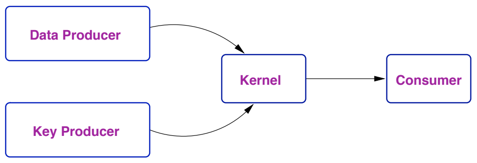
It is through the use of types for interaction, as we shall argue, that we can make the best of this computing model without losing high-level abstraction nor efficiency. For clarity, we consider an idealised model along the lines of [3], where a chip consists of multiple isomorphic VNMs, of the same ISA and each with its own memory. Data sharing is through asynchronous copy of possibly multiple words from one memory to another, or DMA. For simplicity in this abstract we do not take into consideration either the size of local memory or the maximum unit of transfer [3], and consider only a so-called "push" version of DMA (cf. [15]).

A Type-Directed Compilation Framework. One of the key features of CMP 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 [12, 18]. How can we translate these applications to executables for CMP? The basic idea of our approach is to stipulate *typed communicating processes* at an intermediate compilation step, and perform a *type-directed compilation* [13] onto a typed machine language for CMP. Schematically:

$$\text{DSL } (\mathsf{L2}) \xmapsto{\texttt{pi}} \text{typed processes } (\mathsf{L1}) \xmapsto{\texttt{asm}} \text{typed assembly language for CMP } (\mathsf{L0})$$

L0, L1, L2 refer to abstraction levels. L2 is a (type-safe) domain specific language, whose description is compiled into typed communicating processes (L1, which may as well include imperative features). This is further translated into L0, a typed low-level language for asynchronous CMP. We illustrate the key ideas of this approach using a simple example.

Streaming Example We take a simple program for stream cipher [17].



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 [12, 18], which we omit. Our purpose is to translate this program to a type-safe multicore program.

Processes with Session Types We use processes with session types [9, 19] 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 discuss soon. Secondly it offers an expressive *target* language into which we can efficiently and flexibly translate different kinds of high-level programs. Many concurrent and potentially concurrent programs (such as a streaming example above) may be represented as a collection of structured conversations, where we can abstract their structures as types for conversations.

Below we show a simple process representation of the streaming algorithm given above. The kernel initiates a session:

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

The channels d and k are used for Kernel to receive data and keys from Data Producer and Key Producer, respectively, where 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 needed for safe translation into DMA operations, and follows a simple discipline which is statically verifiable). 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 the recursive agent; $\overline{a}(d,k,c)$ is a session initiation which establishes the 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 can be given as follows.

$$\begin{array}{rcl}\text{DataProducer} & \overset{\text{def}}{=} & \mathsf{def}\ \mathrm{P}(d,k,c) = d?();\ d!\langle data\rangle;\mathrm{P}\langle d,k,c\rangle\ \ \mathsf{in}\ \ a(d,k,c).\mathrm{P}\langle d,k,c\rangle \\ \text{Consumer} & \overset{\text{def}}{=} & \mathsf{def}\ \mathrm{C}(d,k,c) = c!\langle\rangle;\ c?(data);\mathrm{C}\langle d,k,c\rangle\ \ \mathsf{in}\ \ a(d,k,c).\mathrm{C}\langle d,k,c\rangle\end{array}$$

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:

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

Above $\mu\mathtt{t}.T$ 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 $\mathtt{t}.d?\,\langle\rangle; d!\,\langle\mathsf{bool}\rangle; \mathtt{t}$. 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.

Type-Directed Compilation Processes with session types are guaranteed to follow rigorous communication structures, given as types. By tracing this session type, we know beforehand what and when processs will send and receive as messages. Using this information, we can replace message passing in typed processes with direct memory write to a multicore chip.

```
main: {
  main: {
    r1 := getIdleCore
    r2 := getIdleCore
    r3 := getIdleCore
    r4 := getIdleCore
    fork dataProducer at r1
    fork keyProducer at r2
    fork kernel at r3
    fork consumer at r4
    yield
  }
}

dataProducer: {
  data: byte [128]
  ack: byte [0]
  main: {
    // produce data
    get ack
    put data in r3.data
    jump main
  }
}
```

```
keyProducer: {
  key: byte [128]
  ack: byte [0]
  main: {
    // produce key
    get ack
    put key in r3.key
    jump main
  }
}

consumer: {
  buf: byte [128]
  ack: byte [0]
  main: {
    get buff
    // consume buf
    put ack in r3.ack
    jump main
  }
}
```

```
kernel: {
  data: byte [128]
  key: byte [128]
  buf: byte [128]
  ackD: byte [0]
  ackK: byte [0]
  ackC: byte [0]
  main: {
    put ackD in r1.ack
    put ackK in r2.ack
    get data; get key
    r5 := 128; jump loop
  }
  loop: {
    when r5 < 0 jump done
    r6 := data[r4]; r7 := key[r4]
    buf[r4] := r7 xor r6;
    jump loop
  }
  done: {
    get ackS
    put sum in r1.arg
    jump main
  }
}
```

Figure 1: L0 code for the stream example

Since our purpose is to have type-safe compilation, we use a typed assembly language [13] targeted at distributed memory CMP and NoC [1, 5], which we call L0 for brevity. L0 is built on top of MIL [20], which in turn is a multi-threaded extension of TAL [13]. Task scheduling is accomplished by loading a program into a core. This includes copying from the main memory the code and the data required for a run of the core, as well as a snapshot of the current register values.

Figure 1 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 writes, where the addresses are shared at the time a thread is launched.

The block associated with identifier main defines a program comprising, in this case, a single basic block, also named main. The program is intended to be uploaded at some core and its execution launched. Cores terminate their execution with a special instruction **yield**, thus joining the pool of available cores. Cores requiring extra workers get hold idle cores by issuing an instruction of the form r_1:= getIdleCore. The first **fork** instruction in main.main copies program dataProducer to the core in register r_1, copies a snapshot of its registers to the target core, and launches the execution of basic block dataProducer.main. Notice that by getting first the number of required cores and then forking the threads we guarantee that each thread knows all other cores (including its own) via registers r_1 to r_4.

The program associated with identifier dataProducer defines a program comprising two buffer declarations (named data and ack) and a basic block (named main). The core running this program writes its data buffer into the kernel's data buffer, but first needs to make sure it can overwrite the latter. We do all this with L0's support for DMA operations. Instruction **get** ack blocks the core until a corresponding **put** instruction is issued, namely via instruction **put** ackD **in** r_1.ack in basic block kernel.done and the data is safely written. After **put**, the producer asynchronously writes its buffer (with **put** data **in** r_3.data), for which the kernel waits with a**get** data instruction.

Program kernel declares three buffers (two incoming, one outgoing), and another three (empty) buffers used for acknowledgements, signals the data and the key producers that the respective buffers can be written, waits for the completion of the write oper-

4

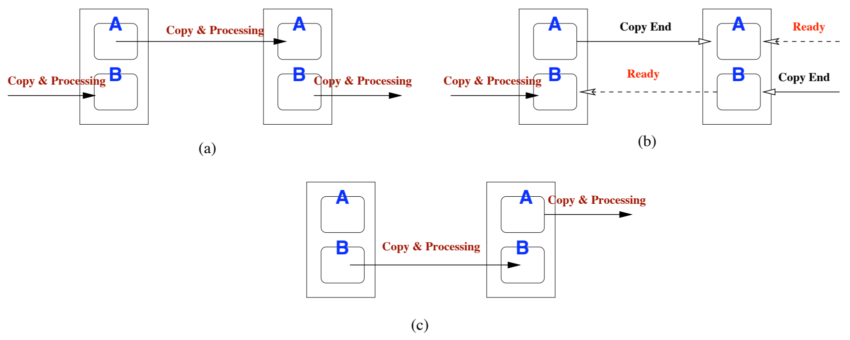


Figure 2: Double-Buffering

ations, and embarks on a loop to fill the outgoing (buf) buffer. Finally it asynchronously writes this buffer into the arg buffer at core consumer before restarting the process. Instruction **when** $r_4 < 0$ **jump** done is expanded into the two instructions $r_4 := r_4 - 1$; **if** $r_4 == 0$ **jump** done, providing for loops.

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 [20] or compare and swap) and DMAs.

Further Topics 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 the same conversations through asynchronous data transfers among local memories of multiple cores, instead of message passing. Processes offer readable, transparent program structures, as well as a target of translation, and types guarantee type-safety of compiled code.

There are several topics which we could not discuss in this abstract. We however briefly touch one topic, which is important for practical implementation. The process-based representation of stream can be made more asynchronous, by transforming the protocol structure slightly. The transformation is simple. 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 "$s \lhd \mathsf{ReadyA}$;" (resp. "$s \rhd \mathsf{ReadyA}$") sends (resp. receives) a signal which tells A is empty.

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

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. This scheme is close to so-called double buffering technique used in multicore processors [10], as shown in Figure 2: indeed, by the same translation scheme, this conversation structure is compiled into a (type-safe) double-buffering implementation of streams, which is much more efficient than the original version due to the exploitation of asynchrony.

This alternative presentation suggests inherent flexibility in compilation and execution of concurrent programs in CMP and other extremely concurrent computing environments, opening new opportunities and challenges. For example we may need more flexibility and generality in type structures (as in the case of, for example, the typing for the process representing double buffering discussed above), new compilation and static analysis techniques, new runtime architectures, and new abstractions. Research from multiple directions (here we only refer to [2, 6, 7] among many closely related and/or complementary works) will be needed to explore this rich field of structured concurrent programming.

References

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

[2] P. Charles, 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*, pages 519–538, 2005.

[3] D. Culler, 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(7):1–12, 1993.

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

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

[6] R. Ennals, R. Sharp, and A. Mycroft. Linear types for packet processing. In *ESOP*, pages 204–218, 2004.

[7] M. Fähndrich, 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, pages 177–190. ACM Press, 2006.

[8] P. Gelsinger, P. Gargini, G. Parker, and A. Yu. Microprocessors circa 2000. *IEEE SPectrum*, pages 43–47, 1989.

[9] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

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

[11] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[12] C.-K. Lin and A. P. Black. DirectFlow: A domain-specific language for information-flow systems. In *ECOOP*, volume 4609 of *LNCS*, pages 299–322. Springer, 2007.

[13] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

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

[15] D. Pham et al. The design and implementation of a first-generation cell processor. In *ISSCC Dig. Tech. Papers*, pages 184–185. IEEE, February 2005.

[16] F. J. Pollack. New microarchitecture challenges in the coming generations of cmos process technologies. In *MICRO*, 1999.

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

[18] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. StreamFlex: high-throughput stream programming in java. In *OOPSLA*, pages 211–228. ACM, 2007.

[19] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.

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