Programming Language Design and Analysis motivated by Hardware Evolution
(Invited Presentation)

Alan Mycroft

Computer Laboratory, University of Cambridge
William Gates Building, JJ Thomson Avenue,
Cambridge CB3 0FD, UK
`http://www.cl.cam.ac.uk/users/am`

Abstract. Silicon chip design has passed a threshold whereby exponentially increasing transistor density (Moore's Law) no longer translates into increased processing power for single-processor architectures. Moore's Law now expresses itself as an exponentially increasing number of processing cores per fixed-size chip.
We survey this process and its implications on programming language design and static analysis. Particular aspects addressed include the reduced reliability of ever-smaller components, the problems of physical distribution of programs and the growing problems of providing shared memory.

1 Hardware Background

Twenty years ago (1985 to be more precise) it was all so easy—processors and *matching* implementation languages were straightforward. The 5-stage pipeline of MIPS or SPARC was well established, the 80386 meant that the x86 architecture was now also 32-bit, and memory (DRAM) took 1–2 cycles to access. Moreover ANSI were in the process of standardising C which provided a near-perfect match to these architectures.

– Each primitive operation in C roughly[1] corresponded to one machine operator and took unit time.
– Virtual memory complicated the picture, but we largely took the view this was an "operating systems" rather than "application programmer" problem.
– C provided a convenient treaty point telling the programmer what the language did and did not guarantee; and a compiler could safely optimise based on this information. Classical dataflow optimisations (e.g. register allocation and global common sub-expression elimination) became common. A GCC port became a 'tick list requirement' for a new processor.

[1] The main wart was that a `struct` of arbitrary size could be copied with an innocent-looking '`=`' assignment.

Moore's Law (the self-fulfilling guide that the number of transistors per unit area doubles every 18 months) continued to apply: a typical 1985 processor had a feature size of $1.5\,\mu$m, today's use 65 nm. The reduction in component size caused consequent (so called 'scaling') changes: speed increased and voltage was reduced. However, the *power* dissipated by a typical $2\,\text{cm}^2$ chip continued to increase—we saw this in the ever-increasing heat sinks on CPUs. Around 2005 it became clear that Moore's law would not continue to apply sensibly to x86 class processors. Going faster just dissipates too much power: the power needed to distribute a synchronous clock increases with clock speed—a typical uni-processor x86-style processor could spend 30% of its power merely doing this. Equally there are speed-of-light issues: even *light in vacuo* takes 100 ps ($\equiv$ 10 GHz) for the round trip across a 15 mm chip; real transistors driving real capacitive wires take far longer. For example the ITRS[2] works on the basis that the delay down 1 mm of copper on-chip wire (111 ps in 2006) will rise to 977 ps by 2013; this represents a cross-chip round-trip of nearly 30 ns—or 75 clock cycles even at a pedestrian 2.5 GHz—on a 15 mm chip. Of course, off chip-access to external memory will be far worse!

However, while Moore's Law cannot continue forever[3] it is still very much active: the architectural design space merely changed to multi-core processors. Instead of fighting technology to build a 5 GHz Pentium we build two or four 2.4 GHz processors ('cores') on a single chip and deem them 'better'. The individual cores shrink, and their largely independent nature means that we need to worry less about cross-the-whole-chip delays. However, making a four-core 2.4 GHz processor be more useful than a single-core 3 GHz processor requires significant program modification (either by programmer or by program analysis and optimisation); this point is a central issue to this paper and we return to it later.

Slightly surprisingly, over the past 20 years, the size of a typical chip has not changed significantly (making a chip much bigger tends to cause an unacceptable increase in manufacturing defects and costs) merely the density of components on it. This is one source of non-uniformity of scaling—and such non-uniformity may favour one architecture over another.

The current state of commercial research-art is Intel's 2006 announcement of their Tera-scale [11] Research Prototype Chips ($2.75\,\text{cm}^2$, operating at 3.1 GHz). Rattner (Intel's CTO) is quoted[4] as saying:

> "...this chip's design consists of 80 tiles laid out in an 8x10 block array. Each tile includes a small core, or compute element, with a simple instruction set for processing floating-point data, ... The tile also includes a router connecting the core to an on-chip network that links all the cores to each other and gives them access to memory.
>
> "The second major innovation is a 20 megabyte SRAM memory chip that is stacked on and bonded to the processor die. Stacking the die

[2] International Technology Roadmap for Semiconductors, `www.itrs.net`

[3] It is hard to see how a computing device can be smaller than an atom.

[4] `http://www.intel.com/pressroom/archive/releases/20060926corp_b.htm`

makes possible thousands of interconnects and provides more than a terabyte-per-second of bandwidth between memory and the cores."

MIT has been a major player in more academic consideration of processor designs which can be used to *tile* a chip. The RAW processor [21] made on-chip latencies visible to the assembly code processor to give predictable behaviour; the recent SCALE processor [2] addresses similar aims: "The Scale project is developing a new all-purpose programmable computing architecture for future system designs. Scale provides efficient support for all kinds of parallelism including data, thread, and instruction-level parallelism."

The RAMP (Research Accelerator for Multiple Processors) consortium describe [24] an FPGA emulator for a range of new processors, and Asanovic et al. [2] summarise the "Landscape of Parallel Computing Research" from both hardware and software archetype (so-called 'dwarfs') perspectives.

1.1 Hidden architectural changes

The evolution, and particularly speed increase, from early single-chip CPUs to modern processors has not happened merely as a result of technology scaling. Much of the speed increase has been achieved (at significant cost in power dissipation) by spending additional transistors on components such as branch-prediction units, multiple-issue units and caches to compensate for non-uniformities in scaling.

The original RISC ('Reduced Instruction Set Computer') design philosophy of throwing away rarely-used instructions to allow faster execution of common instructions became re-written to a revisionist form "make each transistor pay its way in performance terms". Certainly modern processors, particularly the x86, are hardly 'reduced' however, they do largely conform to this revised view. Thus, while originally it might have been seen as RISC (say) "to remove a division instruction limiting the critical timing path to allow the clock-speed to be increased", later this came to be seen as "re-allocating transistors which have little overall performance effect (e.g. division) to rôles which have greater performance impact (e.g. pipeline, caches, branch prediction hardware, etc)".

Another effect is that speed scaling has happened at very different rates. Processor speeds have increased rather faster than off-chip DRAM speeds. Many programmers are unaware that reading main memory on a typical PC takes *hundreds* of processor cycles. Data caches hide this effect for software which behaves 'nicely' which means not accessing memory as randomly as the acronym RAM would suggest. (Interestingly, modern cache designs seem to be starting to fail on the "make each transistor pay its way in performance terms" measure. Caches often exceed 50% of a chip area, but recent measurements show that typically on SPEC benchmarks that 80% of their content is dead data.)

The result is that the performance of the modern processors depends far more critically on the exact instruction mix being executed than was historically the case; gone are the days when a load took 1–2 cycles.

Addressing such issues of timing is also a source of programming language design and program analysis interest. However it may be that this problem might diminish if, once multi-core is fully accepted, the fashion moved towards a greater number of simpler (and hence more predictable) processors rather than fewer complex processors.

1.2 Other Hardware Effects

As mentioned above, technology scaling reduces the size of components and increases switching speed even though voltage is also reduced. Propagating a signal over a wire whose length is in terms of feature size scales roughly (but not quite as well) as the switching speed of components. However, propagating a signal over a wire whose length is in terms of chip size gets exponentially worse in terms of gate delays. This can be counteracted by using larger driver transistors, or more effectively by placing additional gates (buffering) along the path. But doing computation within these buffers adds very little to the total delay. Two slogans can summarise this situation:

– (the expanding universe): communication with neighbouring components scales well, but every year more components appear between you and the edge of the chip and communicating with these requires either exponentially more delay, or exponentially bigger driver transistors. Thus local computation wins over communication.
– (use gzip at sender and gunzip at receiver); it is worth spending increasing amounts of computation to reduce the number of bits being sent because this allows greater delays between each transition in turn which allows smaller transistors to drive the wire.

Once a long wire has gates on it, then we may as well do something useful with them. This fortuitously overlaps the question of how we enable the many processors on a multi-core chip to communicate with one another. The answer to use is some form of *on-chip network*. This can be fast (if big driver transistors are used) and compare favourably with communication achieved by random point-to-point links.

Reducing the feature size of a chip generally requires its operating voltage to be reduced. However, this reduces noise margins and also increases the 'leakage current' (the classical CMOS assumption is that transistors on a chip only consume one unit of energy when switching); below 65 nm or so leakage current can exceed power consumption due to computation. This gives two effects:

– it is beneficial to turn off the power supply to areas of the chip which are not currently active;
– the reduced noise margin increases the error rate on longer wires—sometimes it is only half-jokingly suggested that TCP/IP might be a good protocol for on-chip networks.

A further effect of feature size reduction is that as transistors become smaller, their state (on/off) is expressed as a small number of electrons. This makes the transistors more susceptible to damage by charged particles e.g. cosmic rays, natural radiation. Charged particles may permanently damage a device, but it is far more common for it to produce a *transient event*, also known as Single Event Transient (SET). For example the electrons liberated by a charged particle may enable a transistor whose input demands it be off (non-conducting) to conduct for a short period of time (600 ps is quoted). Some DRAM memory chips ('ECC') are equipped with redundant memory cells and error-detecting or error-correcting codes generated during write cycles and applied during read cycles; similarly aerospace often uses multiple independent devices and majority voting circuits. It is notable that a standard aircraft 'autopilot' function is implemented using five 80386 processors; the larger feature size of the 80386 makes it less susceptible to SET events, and the multiple processors give significant redundancy.

1.3 Summary: Hardware Evolution Effects on Programming

The hardware changes discussed above can be summarised in the following points:

- computation is increasingly cheap compared to communication;
- making effective use of the hardware requires more and more parallelism;
- the idea of global shared memory (and with it notions like semaphores and locking) is becoming less sustainable; on-chip networks increasingly connect components on chip;
- to keep chips cool enough we may have to move CPU-intensive processes around, and (because of leakage current) to disable parts of the chip when they are not being used;
- because of smaller feature sizes, transient hardware errors (leading to errors in data) will become more common.

One additional effect is:

- the time taken for a computation has become much less predictable due to the complexity of uni-processor designs, caches and the like. However, this may reduce in the future were individual processors to become simpler and the idea of a uniform memory space were to be less of a programming language assumption.

The rest of the paper considers programming language consequences.

2 Programming Language Mismatch to Hardware

Most current mainstream programming languages are pretty much stuck in the 1985 model. It is true that C has been superseded by the Object-Oriented paradigm (C++, Java, C#) and this is certainly useful for software engineering as it facilitates larger systems being created by programming teams. However, in many ways very little has changed from C.

Perhaps the critical observation is that the current OO fashion coincided with the period of steady *evolution* of the (single-processor) x86 architecture. The current pressures for *revolutionary* change may (and I argue should) lead to matching language change. Let us examine some particular problems with the OO paradigm (many are inherited from C).

Consider a function (or method) `foo` which takes an element of `class C` as a parameter. In C++, merely to *declare* `foo` (e.g. as part of an interface specification before any code has been written) we have to choose between three possible declarations:

```
extern void foo(C x);
extern void foo(C *x);
extern void foo(C &x);
```

The first one says call-by-value, the last two provide syntactic variants on call-by-reference. While on a single-processor architecture we might discuss these in terms of minor efficiency considerations (e.g. whether `class C` is small enough that its copying involved in call-by-value makes up for additional memory accesses to its fields implied by call-by-reference), on a multi-core architecture the difference is much more fundamental. If call-by-reference is used then the caller of `foo` and the body of `foo` must by executed on processors both of which have access to the memory pointed to by `x`. While call-by-value might therefore seem attractive, when used without care it breaks typical OO assumptions ('object identity'). E.g. toggling one switch twice may well have a different effect from toggling both a switch and its clone.

It might be suggested that Java avoids this issue; however it avoids it in the way that in '1984' Orwell's totalitarian government encourages 'Newspeak' to avoid thought-crime by making it inexpressible in the language. In Java all calls are by reference; therefore caller and callee must execute on processors sharing access to passed data. Call-by-value is at best clumsily expressed via remote method invocation (RMI); moreover, any suggestion that RMI can be introduced "where necessary" to distribute a Java system is doomed to fail due to the very different syntax and semantics of local and remote method invocation.

Incidentally, the `restrict` qualifier found in C99 does not help significantly here in that it keeps the assumption of global shared memory, but merely allows the compiler to make various (non-)aliasing assumptions.

We will return to this topic below, but I would like to argue that it is inappropriate to have interface specification syntax which places restrictions ('early binding') on physical distribution of methods; one suggestion is that of a C++ variant declaration

```
extern void foo(C @x);
```

with meaning "I've not yet decided whether to pass `x` by value or by reference, so fault my program if I do an operation which would have differing semantics under the two regimes."

This would allow *late binding* of physical distribution: all uses of '`@`' can be treated as copy (if caller and callee do not share a memory space at reasonable cost) or by alias (if caller and callee execute on the same address space).

2.1 What Should This Community Learn?

We will return to topics below in more detail, but we list them here to motivate the topics we choose below.

Processor developments expose the fact that C-like languages do not capture important properties of system design and implementation for such architectures. Important issues which we might want to expose to aid writing software for such architectures include:

– late binding of physical distribution;
– more expressive, but concise interface specifications;
– systematic treatment of transient errors.

2.2 Why not stick with C++/Java/C#?

For many traditional applications this will suffice, e.g. editors, compilers, spreadsheets and the like will continue to work well on a single core of a multi-core processor. But current and future applications (e.g. weather forecasting, sophisticated modelling[5]) will continue to demand effective exploitation of hardware—and this means exploiting concurrency.

Remember also that one particular challenge will be to execute programs written for packages—such as Matlab—effectively.

3 Programming Language Design *or* Program Analysis

A great deal of work in this community concerns program *analysis*. While I have personally worked on program analysis, and continue to believe in it for local analysis, I now have significant doubts as regards whole-program (inter-module) analyses and their software engineering impact. The problem is that of *discontinuity*: such a large-scale analysis may enable some optimisation which gains a large speed-up, for example by enabling physical distribution. However, a programmer may then make a seemingly minor change (perhaps one of a long sequence of changes) to a program only to find that it now runs many times slower. This real problem is that it is hard to formulate what the programmer 'did wrong' and to enable understanding of how to make a similar change without suffering this penalty.

Type systems and properties expressed by type-like systems seem to provide a better answer: properties of interest are then explicitly represented within a

[5] One might wonder how *our* community might exploit large-scale concurrency. Do our programs match any of the software dwarfs of Asanovic et al. [2]? Or will we merely continue to use use one core of our multi-core chips as we do at the moment?

programming language at interface points. For example, the interface to a procedure may express the property that its parameter will be consumed (logically de-allocated); then callers of the procedure can check that no use is made of the parameter after the call. This means that violating important assumptions will result in errors which have human-comprehensible explanations.

Of course, there is no problem with *local* inference—a tasteful mechanism always avoids pointless specification—the advantage is that programmer-specified invariants can be used to anchor local reasoning and to break down a global analysis into many smaller independent analyses.

For various aspects of multi-core programming (e.g. running two blocks of code concurrently), it is important to know whether two pointers may alias. Much important work has been done on *alias analysis* (which is undecidable in theory and for which achieving good approximations is problematic in practice). However, we still lack designs for programming languages in which aliasing information can be expressed within the language rather than as a mere analysis.

This situation can be compared with the two ways of adding types to a dynamically-typed language. Method 1 is to analyse the program determining which variables have known type (and optimising their accesses) and which have to adopt a fall-back 'could be anything' treatment. Method 2 is to add a mandatory type system for the language which allows all variable accesses to be optimised and which rejects as few programs as possible. Lisp (with Soft Typing) and ML might serve as good templates here.

An implementation language in which programmer knowledge of known aliasing (and non-aliasing) can be expressed in interface specifications succinctly, and acceptably to an ordinary programmer, would be a particular success here.

4 Programming Languages: High-Level and Implementation Level

Some might be surprised at my focus on C in the introduction, when there were arguably many more 'interesting' language features being explored in 1985. I focused on C because it corresponded directly to hardware features in 1985 (it was a good *implementation* language), and indeed could (and did) usefully serve as an intermediate language for compilers from higher-level languages.

The same motivation was also present in the design of Occam [10] which was a well-matched implementation language for an early multi-core computer[6]—the 'transputer'.[7] Occam was modelled on Hoare's CSP so naturally supported message passing (separate transputers had no shared memory), and represented concurrency directly. Sadly, in many ways it was ahead of its time, and we would have perhaps been in a better position with respect to higher-level language

[6] The 'cores' here were actually separate chips containing 16-bit (later 32-bit) processors with their own memory and four fast I/O ports (e.g. with nearest neighbours on a rectangular grid).

[7] `http://en.wikipedia.org/wiki/INMOS_transputer` seems the best reference nowadays.

for multi-core had more work been done then on novel higher-level features to compile to Occam!

My feeling here is that we need both sorts of language; we need a good implementation language which is well-matched to hardware, and this exhibits the 'sharp end' of many challenges. However, high-level languages may also need to change somewhat, and this is taken up in the next section with "pointers considered harmful" and Section 5.7 explores the desire to have late binding of store-versus-recompute decisions. Some form of structuring beyond "everything is an object (or value) which can be used anywhere at any time" appears likely to be useful.

5 Some Interesting Directions

This section discusses work or possible future projects (inevitably biased towards my own interests) which could be useful in addressing the mismatch between current languages and future hardware.

5.1 (Simple) Pointers Considered Harmful

We have already identified how the ubiquitous use of call-by-reference for objects causes two forms of problems. Firstly in practice it becomes almost impossible to determine whether two object references point to the same object or not, this inhibits parallelisation since code as simple as

```
for (i=0; i<NCHANS; i++) process_channel(i);
```

is often not parallelisable because of the possibility of aliasing of some reference in `process_channel(0)` with a reference in `process_channel(1)`. Secondly, pointers inhibit physical distribution in situations where memory access is not uniform, e.g. if a caller and callee are on different physical processors then we need to ensure that any call-by-reference parameter lives in memory accessible to both—and this may not exist or may be much slower to access than fast local memory. C99's `restrict` qualifier can sometimes help with the former problem but not with the latter.

Shape Analysis [16], Uniqueness types [13] and Ownership Types [3] provide some purchase on this problem. They identify a similar theme: unrestricted pointers are too powerful. In many ways they are like the unrestricted use of labels and `goto`s which Dijkstra railed against in "`goto` considered harmful". Given that a pointer to local data on one processor is not necessarily even valid on another processor, we need some way to tame pointers. This can happen in more than one way: either we wish to control the number of aliases to a given object, or we wish to ascribe an address space to a pointer, so that dereferencing a pointer is only valid on processors which have a capability to do so.

In our work on PacLang [4] we showed how a quasi-linear[8] type system allows one to write code naturally in which an object could move between processors

[8] A linear type system requires each object to have a single active pointer and when that pointer is assigned or passed to a function then it may no longer be used to

with compile time checking of linearity assumptions. A purely linear type system tends to require rather uncomfortable passing of values back and forth; a quasi-linear system enhances this with limited-lifetime second-class pointers which make local call-by-reference possible in a natural programming style. It turns out that linear (or quasi-linear) knowledge of pointers helps [5] in refactoring code from sequential to concurrent by telling a compiler that certain aliasing—which would lead to a race condition—cannot happen. A particular noteworthy point was the concept of an "Architecture Mapping Script (AMS)" which specified architecture details so that late binding of processes to processors could be achieved.

Microsoft's Singularity OS (Fähndrich et al. [6]) project further develops this idea to a message-passing operating system—linear data buffers (allocated in `ExHeap`) can be transferred from process to process by merely passing a pointer since linearity ensures that the sender no longer has access to the data after transfer.

Region-based type systems [19] distinguish pointer types with the region (think 'address space') into which they may legitimately point. They provide a good basis for providing syntax to describe situations in which a pointer to local memory in one processor is being passed via a second processor (on which it is not valid to dereference it) onto a third which can dereference it. However, as Fähndrich et al. observe, the original lexical nesting structure of regions cannot be retained as-is.

5.2 The Actor Model

In the Actor Model of computation, all inter-process communication is achieved by message passing; actors only access disjoint memory. Given that architectural developments mean that communication is becoming the dominant cost rather than computation, actor-based languages are appealing for their explicit representation of non-local communication. A notable commercial example is that of Erlang [1].

5.3 Theoretical Models of Restricted Re-Use

We can see all the above models as attempts to control how and when value might validly be accessed—contrast this with traditional shared memory in which any value may be accessed at any time so long as it has not been overwritten.

While models based on linear types have been mentioned several times above (Wadler [20] is the seminal explanation of this), Separation Logic [14] has recently attracted rapidly growing interest. Separation logic at its simplest expresses assertions that an address space is split into two or more disjoint areas—not

reference the object; only the new copy may be so used. In C++ a dynamic version of this concept is enshrined as the `auto_ptr` class and invalidation of old pointers is achieved by overriding the assignment operator.

only can this model the sort of situations we have seen above, but it can also model dynamic change of ownership and shared-memory systems very effectively.

Both linear types and separation logic provide mechanisms for describing values which are not freely accessible from the whole program. However, they describe overlapping rather than identical phenomena and a formal connection between them would be highly desirable for inspiring work on concise programming language representations of restricted re-use.

5.4 More on Interface specifications

We have already seen that providing interface specification on pointers can provide compile-time to programs to enable them to be better mapped onto multi-core hardware.

However, there are other ways in which interfaces are often inexpressive and we give a hardware example.[9] Consider the Verilog encoding of a two-stage shift register: the input byte on '`in`' appears on the output '`out`' two clock ticks later:

```
module two_stage (in, out, clock);
   input [7:0] in;
   output [7:0] out;
   input clock;

   reg [7:0] state1;
   reg [7:0] state2;
   assign out = state2;         // or out = state1
   always @(posedge clock)
         begin
            state1 <= in;
            state2 <= state1;
         end
endmodule
```

In this example the module specification contains the classical programming language knowledge that `in`, `out`, and `clock` are wires of given width and which of them are outputs. However, suppose we change the commented line to

```
   assign out = state1;
```

then the code behaves as a one-stage delay instead. In an ideal world, we would like this timing information, or indeed information that one signal is only valid after another goes high, to be part of the type information in the module specification so that if part of a circuit is retimed—say to equalise computation

[9] It is admitted that HDLs (Hardware Description Languages) are moving towards using FIFO "channels" to add flexibility in the time domain to ease this sort of problem, particularly for crossing clock domain boundaries, but the example given illustrates another way in which classical programming language "names and types" interface formalism can be seen as lacking.

delays—then type-checking errors can be raised for places in the code which have not taken this retiming into account.

Does such information have a place in future programming languages?

5.5 Fractals in Programming and Architecture

Rent's Rule (Stroobandt [17] gives a good overview) enshrines the empirical observation that the number of pins T on a chip tends to follow a power law $T = T_0 g^p$ where g is the number of internal components (gates) and T_0 and p are constants. Donath noted that Rent's Rule could also be be used to estimate wire-length distribution in VLSI chips.

Power laws tend to suggest that there is an underlying self-similarity (also known as the system being fractal). Of course, the very basis of top-down engineering (software or otherwise) is that each component is hierarchically built of a number of smaller components. This also is self-similar.

It is intriguing to consider whether these two observations could be exploited to improve our understanding of how to map complex systems to hardware. With the exception of references to global memory (heap-allocated data structures—recall the "pointers considered harmful" slogan above) data flow follows design decomposition which is very encouraging. Do global-memory-free programs have a better mapping to hardware? Can such programs be expressive enough?

Of course, global memory breaks this assumption because it provides a way to move data from any part of the system to another in a relatively uncontrolled manner; in general memory access requires some form of serialisation which slows down processing elements. One question is whether designs like Intel's 'Tera-scale' (bonding the memory directly on top of the processor die essentially exploits a scale-free short-path access in the third dimension, see Section 1) will provide enough bandwidth to (and cache-coherency of!) global memory so that shared-memory models are still valid or whether the concept of global memory is still ultimately problematical.

5.6 Limits for Speed-ups

In seminal work Wall [23] analyses instruction traces for various benchmarks including an early version of the SPEC[10] benchmark suite. He calculated limits to speed-up based on *instruction-level* parallelism under various models of behaviour of caches and branch prediction. Redux [12] constructed a more abstract "Dynamic Data Flow Graph (DDFG)" from a computation which reflected merely the computations necessary to compute the result. The depth of the DDFG therefore represents the computation given unbounded parallelism (not necessarily limited to instruction-level parallelism).

An interesting research direction might be to explore more whether DDFGs for various benchmarks might fit (or be made to fit by source-level adjustment) a world in which parallel computation is cheap, but communication is expensive. To what extent to DDFGs express fractal structure inherent in programs?

[10] `www.spec.org`

5.7 Store versus Re-compute

Traditionally, the sequential nature of computers has generally made programmers aware that re-computation takes time and the way to solve this is to store values for later re-use. However, as memory becomes more distributed then local re-computation of some values (especially data structures in memory) may become cheaper than accessing remote memory.

We have already seen a small version of this in optimising compilers: if a small common sub-expression (e.g. `x+1`), which would normally held in a register, has to be spilled to memory, then it is better to *rematerialise* it (i.e. recompute it, thereby undoing the CSE optimisation) rather than accept the cost of a store followed by a re-load.

Again it would be desirable to to have language features which allow programs to be designed and developed more neutrally with respect to store versus recompute ('late binding on the store/recompute axis') than is currently the case. This would also facilitate porting to multiple architectures.

5.8 Opportunistic Concurrency and Related Techniques

While the status of global memory in multi-processor systems is unclear, there is much scope for examining alternatives to today's ubiquitous semaphore-based locking and unlocking mechanisms which can be expensive due both to the cost of memory synchronisation for atomic test-and-set or compare-and-swap instructions and also to the fact that programmers often find it easier to take a coarse-grain lock instead of reasoning about the correctness of fine-grain locking.

One of these is Software Transactional Memory (STM) [7], in which lock and unlock are replaced by an `atomic` block. Atomic blocks execute speculatively in that either they execute to completion without interference from other processes able to access given memory, or such interference is detected and the atomic block (repeatedly) re-tried. This idea is already familiar from databases. Software Transactional Memory represents an interesting mid-point between the notion of lockable global memory and the notion of message-passing.

Worth grouping with STM is Rundberg and Stenstrom's Data Dependence Speculation System [15] in which ambiguities in compile-time alias analysis are resolved at run time by speculatively executing threads concurrently but with a dynamic test which effectively suppresses writes occurring out of order and restarts the threads from a suitable point.

Lokhmotov et al. [9] describe Codeplay's *sieve* construct which side-steps many of the problems with alias analysis. A *sieve* block has writes with the block delayed until the block exit. This allows a programmer to express the absence of read-after-write dependencies (which often have to be assumed present due to inaccuracy of alias analysis); the effect is to allow C-like code to be optimised into a "DMA-in, process in parallel, then DMA-out" form.

Finally, there has been significant work on pre-fetch- or turbo-threads in which two versions of code are compiled. The second one is the normal code, and the first one is a cut-down version which is intended to execute in advance of the second one; it does reduced computation with no stores to global memory and all loads from local memory replaced by pre-fetch instructions (start load into cache but do not wait for result).

5.9 Programming against Hazards and Transient Errors

This is a relatively new topic for our community. We probably are all aware that highly safety-critical (usually for aerospace applications) electronics is often duplicated and a majority-voting circuit avoids any single processor crashing (temporarily or even permanently) causing mission failure. Embedded systems programmers have for years used 'watchdog timers' to ensure that systems suffering temporary failure (e.g. an infinite loop caused by data being corrupted by a cosmic ray), can be restarted relatively quickly. The idea is that the 'watchdog' has to be sent a message periodically confirming the sender is still alive; if this does not happen then the watchdog (hardware) timer causes a system reset. Of course, there have been research groups focusing on overall system reliability for decades, but recently there has been several novel approaches which intersect our community's interests more directly.

Some work which springs to mind includes:

- Sarah Thompson's thesis [18] showing not only that hazards (narrow pulses on an otherwise clean signal or transition) can be modelled by abstract interpretation, but also that majority voting circuits are theoretically incapable of removing them without resorting to timing.
- Walker et al. describe lambda-zap [22] whose reduction both inserts faults and also duplicates computations to be resolved by majority voting. There is also a type system that guarantees well-typed programs can tolerate a single error.
- Hillston's PEPA [8] (Performance Evaluation Process Algebra) which can model quantitative aspects of systems, originally reaction rates, but hopefully also failure rates.

Finally, there is idea that performance and reliability can be traded, for example we might eventually require some form of redundant computation—perhaps merely at the hardware level—to give enough reliability for (say) a spreadsheet, but be tolerant of errors during some applications (e.g. rendering graphics for display where the human eye either would not notice errors or would ignore them).

6 Conclusions

We have seen that forces in hardware design are increasing the mismatch between traditional programming languages and future processor designs. I argue that *languages* have to evolve to take into account the new emphasis on concurrency and the reduced ability to view an object as a simple pointer—and more expressive interface specifications are pivotal. Program *analysis* techniques can provide inspiration for these future designs.

In Addition, there remains scope for language features designed to address issues directly concerning hardware; for example, how is unreliability best expressed? Can we improve the expressivity of hardware and software interfaces to document better their behaviour?

In the bigger picture, there is still much scope for higher-level language designs which encourage programmers to think in a way which naturally encodes effectively on coming architectures—and even for new architectural features corresponding to programming innovations. Can we return to the comfort of 1985 when implementation languages and computer architecture *matched*?

Acknowledgements

Thanks are due to Simon Moore at Cambridge for many useful ideas and comments.

References

1. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang.* Prentice Hall, 1996.
2. K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: a view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 18 2006.
3. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 48–64, New York, NY, USA, 1998. ACM Press.
4. R. Ennals, R. Sharp, and A. Mycroft. Linear types for packet processing. In D. A. Schmidt, editor, *European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2004.
5. R. Ennals, R. Sharp, and A. Mycroft. Task partitioning for multi-core network processors. In R. Bodík, editor, *Compiler Construction*, volume 3443 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2005.
6. M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity OS. In *EuroSys '06: Proceedings of the 2006 EuroSys conference*, pages 177–190, New York, NY, USA, 2006. ACM Press.
7. T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, volume 38, pages 388–402, New York, NY, USA, November 2003. ACM Press.
8. J. Hillston. Tuning systems: from composition to performance The Needham Lecture. *Comput. J.*, 48(4):385–400, 2005.

9. A. Lokhmotov, A. Mycroft, and A. Richards. Delayed side-effects ease multi-core programming. In *Euro-Par 2007 Parallel Processing*, Lecture Notes in Computer Science. Springer, 2007.
10. INMOS Ltd. *OCCAM Programming Manual.* Prentice-Hall International, London, 1984.
11. Intel Corporation. Terascale computing.
`http://www.intel.com/research/platform/terascale/index.htm`
12. N. Nethercote and A. Mycroft. Redux: A dynamic dataflow tracer. *Electr. Notes Theor. Comput. Sci.*, 89(2), 2003.
13. M. Plasmeijer and M. van Eekelen. Language report: concurrent Clean, 1998. Technical Report CSI-R9816, Computing Science Institute, University of Nijmegen, The Netherlands.
14. J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.
15. P. Rundberg and P. Stenstrom. An All-Software Thread-Level Data Dependence Speculation System for Multiprocessors. *The Journal of Instruction-Level Parallelism*, 1999.
16. S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, pages 105–118, 1999.
17. D. Stroobandt. Recent advances in system-level interconnect prediction. *IEEE Circuits and Systems Society Newsletter*, 11(4):4–20, 2000.
Available at `http://www.nd.edu/~stjoseph/newscas/`
18. S. Thompson. On the application of program analysis and transformation to high reliability hardware. Technical Report UCAM-CL-TR-670 (PhD thesis), University of Cambridge, Computer Laboratory, July 2006.
Available at `http://www.cl.cam.ac.uk/techreports`
19. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
20. P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.
21. E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, 1997.
22. D. Walker, L. Mackey, J. Ligatti, G. Reis, and D. August. Static typing for a faulty lambda calculus. In *ACM International Conference on Functional Programming*, Portland, Sept. 2006.
23. D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 26, pages 176–189, New York, NY, 1991. ACM Press.
24. J. Wawrzynek, M. Oskin, C. Kozyrakis, D. Chiou, D. A. Patterson, S.-L. Lu, J. C. Hoe, and K. Asanovic. Ramp: a research accelerator for multiple processors. Technical Report UCB/EECS-2006-158, EECS Department, University of California, Berkeley, November 24 2006.