acmqueue

No Such Thing as a General-purpose Processor

And the belief in such a device is harmful

David Chisnall, University of Cambridge

There is an increasing trend in computer architecture to categorize processors and accelerators as "general purpose." Of the papers published at this year's International Symposium on Computer Architecture (ISCA 2014), nine out of 45 explicitly referred to general-purpose processors; one additionally referred to general-purpose FPGAs (field-programmable gate arrays), and another referred to general-purpose MIMD (multiple instruction, multiple data) supercomputers, stretching the definition to the breaking point. This article presents the argument that there is no such thing as a truly general-purpose processor and that the belief in such a device is harmful.

TURING COMPLETENESS

Many of the papers presented at ISCA 2014 that did not explicitly refer to general-purpose processors or cores did instead refer to general-purpose programs, typically in the context of a GPGPU (general-purpose graphics processing unit), a term with an inherent contradiction.

A modern GPU has I/O facilities, can run programs of arbitrary sizes (or, if not, can store temporary results and start a new program phase), supports a wide range of arithmetic, has complex flow control, and so on. Implementing Conway's Game of Life on a GPU is a fairly common exercise for students, so it's clear that the underlying substrate is Turing complete.

Here is one definition of a general-purpose processor: if it can run any algorithm, then it is general purpose. This is not a particularly interesting definition, because it ignores the performance aspect that has been the driving goal for most processor development.

It's therefore not enough for a processor to be Turing complete in order to be classified as general purpose; it must be able to run all programs *efficiently*. The existence of accelerators (including GPUs) indicates that all attempts thus far at building a general-purpose processor have failed. If they had succeeded, then they would be efficient at running the algorithms delegated to accelerators, and there would be no market for accelerators.

With this in mind, let's explore what people really mean when they refer to a general-purpose processor: the specific category of workloads that these devices are optimized for and what those optimizations are.

THE OPERATING SYSTEM

A common requirement for a general-purpose processor is that it can run a general-purpose operating system—meaning an operating system that is either Unix or (like MS Windows) has a similar set of underlying abstractions to Unix. For example, most modern processors lack the ability to cleanly express the memory model found in Multics, with fine-grained sharing and transparent virtualized access to memory-mapped I/O devices.

The ability to run an operating system is fundamental to the accepted definition. If you remove

the ability to run an operating system from a processor that is considered general purpose, then the result is usually described as a microcontroller. Some devices that are now regarded as microcontrollers were considered general-purpose CPUs before the ability to run a multitasking, protected-mode operating system became a core requirement.

The important requirements for running an operating system (again, with the caveat that it must be efficient and not simply running an emulator) are similar to those enumerated by Gerald Popek and Robert Goldberg for virtualization.[6] This is not surprising, as an operating-system process is effectively a virtual machine, albeit with a very high-level set of virtual devices (file systems rather than disks, sockets rather than Ethernet controllers).

To meet these requirements, a processor needs to distinguish between privileged and unprivileged instructions and the ability to trap into privileged mode when a privileged instruction is executed in unprivileged mode. Additionally, it needs to implement protected memory so that unprivileged programs can be isolated from each other. This is somewhat weaker than Popek and Goldberg's requirement that memory accesses be virtualized, because an operating system does not necessarily have to give every process the illusion that it has an isolated linear address space.

Traditionally, all I/O-related instructions were privileged. One of the advantages of memory-mapped I/O regions, as opposed to separate namespaces for I/O ports, is that it makes it possible to delegate direct access to hardware to unprivileged programs. This is increasingly common with network interfaces and GPUs but requires that the hardware provide some virtualization support (e.g., isolated command queues that cannot request direct memory access to or from memory controlled by another program).

In MIPS, the privileged instructions are modeled as being implemented by CP0 (control coprocessor). This dates back to the pre-microcomputer model where the CPU consisted of a small number of chips and the MMU (memory management unit) was usually placed on a different chip (or even collection of chips).

In a modern multicore system, it's fairly common to allocate some cores almost purely to user-space applications, with all interrupts routed away from them so that they can run until a scheduler on another core determines that they should stop. It's therefore not clear whether every processor in a multicore system needs to be able to run all of the kernel code. This model was explored in the Cell microprocessor,[3] where each SPE (synergistic processing element) was able to run a small stub that would message the PowerPC core when it needed operating-system functionality. This design is relatively common in supercomputers, where each node contains a slow commodity processor to run an operating system and handle I/O, as well as a much faster processor optimized for HPC (high-performance computing) tasks, over which the application has exclusive control.

The Barrelfish research operating system[1] proposes that, as the transition is made from multicore to many-core architectures, time-division multiplexing makes less sense for virtualization than space-division multiplexing. Rather than one core running multiple applications, one application will have exclusive use of one or more cores for as long as it runs. If this transition takes place, it will change the requirements considerably.

BRANCHING

The heuristic used in the RISC I processor,[5] discovered by examining the code produced by the Portable C Compiler on the Unix code base, was that general-purpose code contains approximately

one branch instruction for every seven instructions. This observation indicates why branch predictors are such an important feature in most modern superscalar pipelined CPUs. The Pentium 4, for example, could have around 140 instructions in-flight at a time and thus (if this observation is correct) needed to predict correctly 20 branches into the future to keep the pipeline full. With a 90 percent branch predictor hit rate, it would have only a 12 percent probability of correctly predicting the next 20 branches.

Fortunately for the Pentium 4, not all branches are of equal difficulty and the most common branch in CPU-dependent code is the backwards branch at the end of a loop. The Pentium 4 included a static heuristic that this branch is always taken on first use, and a complex loop predictor would determine which iteration would be the last one.

This observation is still largely true for C code. It is also now largely true for other languages but with several caveats. In particular, while it is true for naïve compilation of C, it is definitely not true for naïve compilation of dynamic languages such as Smalltalk or JavaScript.

Consider the simple expression *a*+*b*. In C, you statically know the types of both *a* and *b*. The operation is either an integer or floating-point addition. Now consider the same line of code in C++. If *a* is a primitive type, then this is a simple arithmetic operation (although there may be some complex code including function calls to coerce *b* to the same type). If *a* is a class that has a virtual overload for the addition operator, then it is an indirect branch via a vtable (virtual table) lookup. The compiler will insert a load of the function pointer at a fixed offset in the vtable, followed by an indirect jump to the loaded address.

In JavaScript, this is even more complex. The exact semantics for addition in JavaScript are quite convoluted; let's simplify by treating all non-numeric addition as equivalent. In the numeric case, all values in JavaScript are double-precision floating-point values. Most processors, however, are much faster when performing integer arithmetic than floating point, so, ideally, all arithmetic that fits into a 53-bit integer (the mantissa size for a 64-bit IEEE-compliant double) would be performed as integer operations. Unfortunately, to do this you must first check whether the operand is an integer or a floating-point value (which adds another branch and typically costs more than the savings obtained from doing the integer operation). You must then check whether it's something else (an object) before finally doing the arithmetic.

This means that in naïvely compiled JavaScript, a simple addition can involve two branches. Optimizing this has been the focus of a huge amount of research over the past 40 years (most of the techniques apply to Smalltalk and similar languages, predating JavaScript). These techniques involve dynamically recompiling the code based on type information gained at runtime; this ensures that the case involving the most common types contains no branches.

It's worth noting that SPARC architecture has a *tagged integer* type and add and subtract instructions specifically designed for dynamic languages.[7] A SPARC tagged integer is 30 bits, with the lowest two bits in a 32-bit word reserved for type information. The tagged operations set a condition code (or trap) if either of the operands has a nonzero tag or if the result overflows.

These instructions are not widely used for several reasons. They use the opposite definitions of tag values from most 32-bit implementations (it is simpler to use a 0 tag for pointers, as it allows the pointer value to be used unmodified). For JavaScript, it is also common to use NaN (not a number) representations to differentiate object pointers from numeric values. This is possible because most 64-bit architectures have a "memory hole"—a range in the middle of the virtual address space that

cannot be mapped—and this conveniently lines up with the pointer interpretation of certain NaN values. This allows generated code to assume that values are floating point and either branch or trap for NaN values.

In spite of the popularity of JavaScript as a general-purpose programming language, most "general-purpose" processors still require the compiler to perform complex convolutions to generate moderately efficient code. Importantly, all of the techniques that do generate good code from JavaScript require a virtual machine that performs dynamic recompilation. If the most efficient way of running general-purpose code on a processor is to implement an emulator for a different (imaginary) architecture, then it is hard to argue that the underlying substrate is truly general purpose.

LOCALITY OF REFERENCE

One feature shared by most modern CPUs is their large caches. In the case of Itanium and POWER, this trend has gone beyond the realm of propriety, but even commodity x86 chips now have more cache than a 386 or 486 had RAM.

Cache, unlike scratchpad memories, is (as its name implies) hidden from the software and exists to speed up access to main memory. This ability to stay hidden is possible only if the software has a predictable access pattern. The strategy most commonly used for CPU caches optimizes for *locality of reference*. The caches are split into lines of typically 64 or 128 bytes. Loading a single byte from main memory will fill an entire cache line, making accesses to adjacent memory cheap. Often, caches will fill more than one adjacent line on misses, further optimizing for this case.

This design is a result of the *working-set hypothesis*, which proposes that each phase of computation accesses a subset of the total memory available to a program, and this set changes relatively slowly.[2] GPUs, in contrast, are optimized for a very different set of algorithms. The memory controller on an Nvidia Tesla GPU, for example, has a small number of fixed access patterns (including some recursive Z-shapes, commonly used for storing volumetric data); it fetches data in those patterns within a texture and streams it to the processing elements. Modern Intel CPUs can also identify some regular access patterns and efficiently stream memory but with a lower throughput.

Jae Min Kim et al. showed that some programs run faster on the slow cores in ARM's big.LITTLE architecture.[4] Their explanation was that the slow Cortex A7 cores have a single-cycle access to their level-1 caches, whereas the fast Cortex A15 cores have a four-cycle penalty. This means that, if the working-set hypothesis holds—the working set fits into the L1 cache, and the performance is dominated by memory access—then the A7 will be able to saturate its pipeline, whereas the A15 will spend most of its time waiting for data.

This highlights the fact that, even within commodity microprocessors from the same manufacturer and generation, the different cache topologies can bias performance in favor of a specific category of algorithm.

MODELS OF PARALLELISM

Parallelism in software comes in a variety of forms and granularity. The most important form for most CPUs is ILP (instruction-level parallelism). Superscalar architectures are specifically designed to take advantage of ILP. They translate the architectural instruction encodings into something more akin to a static single assignment form (ironically, the compiler spends a lot of effort translating from

such a form into a finite-register encoding) so that they can identify independent instructions and dispatch them in parallel.

Although the theoretical limit for ILP can be very high, as much as 150 independent instructions,[8] the amount that it is practical to extract is significantly lower. VLIW (very long instruction word) processors simplify their logic by making it the compiler's job to identify ILP and bundle instructions.

The problem with the VLIW approach is that ILP is a dynamic property. Recall that the RISC I heuristic was that branches occur, on average, every seven instructions. This means that the compiler can, at most, provide seven-way ILP, because it cannot identify ILP that spans basic blocks: if the compiler statically knew the target of a branch, then it would most likely not insert a branch.

VLIW processors have been successful as DSPs (digital signal processors) and GPUs but not in the kind of code for which "general-purpose" processors are optimized, in spite of several attempts by Intel (i860, Itanium). This means that *statically predictable* ILP is relatively low in the code that these processors are expected to run. Superscalar processors do not have the same limitation because, in conjunction with a branch predictor, they can extract ILP from dynamic flow control spanning basic blocks and even across functions.

It's worth noting that, in spite of occupying four times the die area and consuming four times the power, clock-for-clock the ARM Cortex A15 (three-issue, superscalar, out-of-order) achieves only 75-100 percent more performance than the (two-issue, in-order) A7, in spite of being able (theoretically) to exploit a lot more ILP.

The other implicit assumption with regard to parallelism in most CPUs is that communication among parallel threads of execution is both rare and implicit. The latter attribute comes from the C shared-everything model of concurrency, where the only way for threads to communicate is by modifying some shared state. A large amount of logic is required in *cache coherency* to make this efficient, yet it is relevant only for shared-memory parallelism.

Implementing message passing (as embodied both by early languages such as Occam and Actor Smalltalk and by newer ones such as Erlang and Go) on such processors typically involves algorithms that bounce cache-line ownership between cores and involve large amounts of bus traffic.

CONCLUSION

The general-purpose processors of today are highly specialized and designed for running applications compiled from low-level C-like languages. They are virtualized using time-division multiplexing, contain mostly predictable branches roughly every seven instructions, and exhibit a high degree of locality of reference and a low degree of fine-grained parallelism. Although this describes a lot of programs, it is by no means exhaustive.

Because processors optimized for these cases have been the cheapest processing elements that consumers can buy, many algorithms have been coerced into exhibiting some of these properties. With the appearance of cheap programmable GPUs, this has started to change, and naturally data-parallel algorithms are increasingly run on GPUs. Now that GPUs are cheaper per FLOPS (floating-point operations per second) than CPUs, the trend is increasingly toward coercing algorithms that are not naturally data parallel to run on GPUs.

The problem of dark silicon (the portion of a chip that must be left unpowered) means that it is going to be increasingly viable to have lots of different cores on the same die, as long as most of

them are not constantly powered. Efficient designs in such a world will require admitting that there is no one-size-fits-all processor design and that there is a large spectrum, with different trade-offs at different points.

REFERENCES

1. Baumann, A., Barham, P., Dagand, P.-E., Harris, T. L., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., Singhania, A. 2009. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*: 29-44.
2. Denning, P. J. 1968. The working-set model for program behavior. *Communications of the ACM* 11(5): 323-333.
3. Gschwind, M., Hofstee, H. P., Flachs, B., Hopkins, M., Watanabe, Y., Yamazaki, T. 2006. Synergistic processing in Cell's multicore architecture. *IEEE Micro* 26(2): 10-24.
4. Kim, J. M., Seo, S. K., Chung, S. W. 2014. Looking into heterogeneity: when simple is faster. In *The 2nd International Workshop on Parallelism in Mobile Platforms*.
5. Patterson, D. A., Sequin, C. H. 1981. RISC I: a reduced instruction set VLSI computer. In *Proceedings of the 8th Annual Symposium on Computer Architecture*: 443-457.
6. Popek, G. J., Goldberg, R. P. 1974. Formal requirements for virtualizable third-generation architectures. *Communications of the ACM* 17(7): 412-421.
7. SPARC International Inc. 1992. *SPARC Architecture Manual: Version 8*. Upper Saddle River, NJ: Prentice-Hall Inc.
8. Wall, D. W. 1993. Limits of instruction-level parallelism. Technical report, Digital Equipment Corporation Western Research Laboratory.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

DAVID CHISNALL is a researcher at the University of Cambridge, where he works on programming-language design and implementation. He spent several years consulting in between finishing his Ph.D. and arriving at Cambridge, during which time he also wrote books on Xen and the Objective-C and Go programming languages, as well as numerous articles. He also contributes to the LLVM, Clang, FreeBSD, GNUstep, and Étoilé open source projects, and he dances the Argentine tango.