Hardware Polymorphism

Adrian Kosmaczewski

2006-04-08

Since data and instructions are stored in RAM in pretty much the same way, a priori the CPU cannot distinguish each other, but by the cycle in which the binary chunk is fetched from memory. In the case of instructions, it then needs to decode the operation codes into instructions, with the added problem that if the operation is performed on data that is not implied by the operation code, the results are wrong or even catastrophic.

The question is: would it be useful if in hardware, each cell of data would carry its own type designation? I will discuss here the pros and cons of this approach, in respect to hardware and software architectures.

Introduction

The example of the + sign is particularly interesting; here's an excerpt of a tutorial for the Ruby programming language, where the need for data types appears in a straightforward way:

"Before we get any further, we should make sure we understand the difference between numbers and digits. 12 is a number, but '12' is a string of two digits.

Let's play around with this for a while:

```
puts  12  +  12
puts '12' + '12'
puts '12  +  12'
```

(results)

```
24
1212
12  +  12
```

How about this:

```
puts  2  *  5
puts '2' *  5
puts '2  *  5'
```

(results)

```
10
22222
2  *  5
```

(Chris Pine, 2006)

As we can see in the above example, higher-level programming languages allow us to distinguish (if not explicitly like Java, contextually like Ruby) among different types of information, whereas, at hardware level, this distinction does not exist; the processor executes or processes instructions or data depending on the processor cycle.

Metadata

In other words, if meaningful information (data) is stored in the memory of the computer and can be processed by a computer program, then we can say that every bit (no pun intended) of information has, at least at a certain abstraction level, and from a certain point of view, a particular type, or, more generally, some metadata attached to it:

“Metadata (Greek: meta- + Latin: data”information”), literally “data about data”, is information that describes another set of data. A common example is a library catalog card, which contains data about the contents and location of a book: It is data about the data in the book referred to by the card. Other common contents of metadata include the source or author of the described dataset, how it should be accessed, and its limitations.”

(Wikipedia, 2006)

In our case, the type of a particular piece of data is part of its metadata:

“Assigning datatypes (”typing”) has the basic purpose of giving some semantic meaning to otherwise meaningless collections of bits.”

(Wikipedia, 2006)

(Imaginary) Type-checking processor architecture

Now, let’s suppose that a certain processor architecture allows us to distinguish in-memory pieces of data from in-memory program instructions. How could this be implemented?

First of all, let’s see the different primitive types of information that a processor could distinguish:

- Integer numbers (of different but defined lengths such as 8, 16, 32 or 64 bits)
- Floating-point numbers (again, of different but defined lengths)
- Single characters (single- or multibyte-characters, such as Unicode ones)
- Strings (of variable lengths)

- Pure binary streams (images, audio, video)
- Uniform arrays or vectors (of variable length, where the items are all of the same type)
- Variable arrays or vectors (of variable length, where the individual items can be of any type, similar to C structures)

Let's imagine, to begin, that this is the definitive list of supported types at hardware level, by a certain microprocessor architecture. I have kept this list particularly close to that of any common high-level language, for reasons that will become obvious in a while.

How could the type metadata be stored at hardware level? The easiest way to imagine this is having a supplemental byte at the beginning of each in-memory variable or structure, indicating the type. This would give us the possibility of referencing 256 different types of data, which is more than enough in this particular example.

In the case of variable length data types as shown above (String, Arrays) another byte (or bytes) should indicate the length of the whole data structure. The need for this will be explained below.

Type-checking

Now, during the execution type, the processor would fetch data from memory but this time, it would have a first byte of information about the information (metadata) indicating the type of what follows, and eventually some length information as well. Instead of having to rely in context (which is the case by now), the processor could proactively check that the data will be processed by the appropriate instructions. This is a common technique used in programming languages called "Type Checking":

"The process of verifying and enforcing the constraints of types - type checking - may occur either at compile-time (a static check) or run-time (a dynamic check). Static type-checking becomes a primary task of the semantic analysis carried out by a compiler. If a language enforces type rules strongly (that is, generally allowing only those automatic type conversions which do not lose information), one can refer to the process as strongly typed, if not, as weakly typed."

(Wikipedia, 2006)

In the case of a processor-based type check, it would be a pure strong, dynamic one.

Of course, this introduces the first drawback of this approach; while current processor architectures bypass this check and blindly trust the "context coherence" between data and instruction, the processor would have to execute an internal check to verify them prior to executing the instruction. A smarter approach would be to have the processor to "trust" some executing code, if it comes from a statically-typed compiler, for example; this way, the processor would not execute the type check, and would have a similar behavior as that from current systems; however, it would execute a supplemental check for code coming from dynamically-typed languages (such as scripting languages).

Security

A direct benefit of type-checking processor architectures such as the one described above has to do with security. One of the most common security problems in software today is inherent to the Von Neumann architecture, in which data and instructions are both loaded in memory and share adjacent locations. This security problem is known as "Buffer Overrun" or "Stack Overrun":

"A stack-based buffer overrun occurs when a buffer declared on the stack is overwritten by copying data larger than the buffer. Variables declared on the stack are located next to the return address for the function's caller. The usual culprit is unchecked user input passed to a function such as strcpy, and the result is that the return address for the function gets overwritten by an address chosen by the attacker. In a normal attack, the attacker can get a program with a buffer overrun to do something he considers useful, such as binding a command shell to the port of their choice".

(Howard & LeBlanc, 2003, page 129)

(Howard & LeBlanc follow this statement with a C program that shows the security failure, and explain how it might be exploited to inject code in the computer program and change its behavior)

In the case of the stack buffer overrun, the problem is not only that current processors do not check the type of the data to process, but they do not even check the length of it. Length-checks should be then the first check that a processor should do before processing in-memory data of variable length (as stated above, Strings, Arrays and structures like C structs fall in this category).

This leads to infere that a type-checker processor would be particularly useful in security intensive environments (such as nuclear plants, life support systems, etc) where the tradeoff of performance for an additional security type check could be highly desirable.

Virtual Machines

Virtual machines such as Java or .NET's Common Language Runtime (CLR) both perform length and type checks; in those cases, the virtual machine specification ensure that the code being executed does not perform illegal memory accesses or reference objects of the wrong type at the wrong moment.

For example, .NET's CLR includes a runtime security engine that constantly checks code metadata, to know whether the method calls are trusted or not:

"Permission demands propagate up the stack. When a method call demands a particular type of permission, the security engine must affirm that every component in the stack (prior to the point of the permission demand) has appropriate permissions. If any component does not, the permission demand fails and an exception is thrown to signify this failure. Each frame of the stack can modify the effective set of permissions by calling Assert, Deny or PermitOnly before making calls, and there are also calls to Revert changes made earlier. Taken together, this mechanism results in aggregate behavior that is constrained by the least privileged component that is participating in a given stack region"

(Stutz, Neward & Shilling, 2003, page 185)

This, among other reasons (such as automatic memory management) make virtual machines a "hot topic" in computing nowadays, since they allow to develop much more secure systems, with greater productivity, with fewer resources.

Of course, it must be said, not all security problems have disappeared with virtual machines, but that's another topic.

Other benefits

I think that another performance benefit could come from the fact of having native string manipulation at hardware level. String manipulation is by far the most common operation performed in high level programming languages (where Perl and Basic are the most common examples), but yet until now text strings as such exist only at that high level (and even until recently, when the Standard Type Library appeared, C++ did not even have a native string type - http://www.bgsu.edu/departments/compsci/docs/string.html).

Common string operations that could be implemented at hardware level include string copying, string concatenation and splitting; this way, getting the length or a defined substring of a given string would need a single processor instruction, instead of the current procedures, that imply memory allocation and copying, both extremely expensive in time and resources:

"In all cases, these functions consist of copying all or a subset of a string to another string. The specific steps are:

- Determine the number of characters to copy - Allocate space for the characters - Copy the characters to the new string Because of the memory allocation and copying operations involved, extracting sub-strings is also an expensive operation."

(VBIP.com, 2006)

Conclusion

The inclusion of high-level instructions in processors is not something new. It allows to boost the speed of hardware architectures, providing common operations to be performed at maximum speed at the lowest system level. One good example of existing implementations is the Velocity Engine existing in PowerPC G4 and G5 microprocessors:

"The Velocity Engine, embodied in the G4 and G5 processors, expands the current PowerPC architecture through addition of a 128-bit vector execution unit that operates concurrently with existing integer and floating-point units. This provides for highly parallel operations, allowing for simultaneous execution of up to 16 operations in a single clock cycle. This new approach expands the processor's capabilities to concurrently address high-bandwidth data processing (such as streaming video) and the algorithmic intensive computations which today are handled off-chip by other devices, such as graphics, audio, and modem functions.The AltiVec instruction set allows operation on multiple bits within the 128-bit wide registers. This combination of new instructions, operation in parallel on multiple bits, and wider registers, provide speed enhancements of up to 30x on operations that are common in media processing"

(Apple Computer, 2006)

Some example code in C that uses the AltiVec instruction set is shown in http://developer.apple.com/hardware/ve/tutorial.html

Of course, these implementations greatly impact compiler and operating systems design, but they do not (I think) impact higher-level languages such as Java, C#, or scripting languages such as Perl or Ruby, who tend to be rather platform-independent (both software and hardware).

References

Apple Computer, "Velocity Engine" [Internet], http://developer.apple.com/hardware/ve/ (Accessed February 3rd, 2006)

Chris Pine, "Learn to Program" [Internet], http://pine.fm/LearnToProgram/?Chapter=02 (Accessed February 3rd, 2006)

David Stutz, Ted Neward & Geoff Shilling, "Shared Source CLI Essentials", ISBN 0-596-00351-X, O'Reilly, 2003

Michael Howard & David LeBlanc, "Writing Secure Code, 2nd Edition", ISBN 0-7356-1722-8, Microsoft Press, 2003

VBIP.com, "String Operations" [Internet], http://www.vbip.com/books/1861007302/chapter_7302_04.asp (Accessed February 3rd, 2006)

Wikipedia, "Datatype" [Internet], http://en.wikipedia.org/wiki/Datatype (Accessed February 3rd, 2006)

Wikipedia, "Metadata" [Internet], http://en.wikipedia.org/wiki/Metadata (Accessed February 3rd, 2006)