The SURE Architecture

David May: December 11, 2016

Background

Computer programming is changing. Object-oriented languages, functional languages and others have accelerated software development. But these languages rely on automatic memory management which imposes high overheads when implemented on conventional computer architectures. It seems timely to investigate whether a change in architecture is now needed - similar to the shift from CISC to RISC in the 1980s in response to the shift from assembly programming to high-level languages.

Along with the change in programming languages, there has been a change in computer systems and their applications. Increasingly, computers are communicating with each other and with users, or they are interacting with a physical environment. This gives rise to a need for trust, achieved via an appropriate combination of simplicity, verification, testing and error containment. Again, it seems timely to investigate whether a change in architecture is needed.

An experimental architecture

The architecture described here employs a novel technique for allocating memory; a hardware *garbage collector* is embedded in the processor sharing memory cycles with the processor. The garbage collector runs continually and is normally fast enough to retrieve unused memory space as fast as new space is allocated.

It also checks that memory accesses are valid; dangling pointers and buffer overflows are detected by hardware. An address consists of two parts. One specifies a memory region; the other specifies a location within the region. For a memory access operation, the processor checks that one operand is an address and the other is a value to be used as an offset. It also checks that the offset specified in a memory access instruction lies within the region specified by the address operand.

The principles can be used in conjunction with any processor architecture; it would be simple to create an Intel, ARM, MIPS, SPARC or RISC-V processor including the memory management system described here. There would be a small cost in the hardware needed for the garbage collector; for a 32-bit architecture it needs about 400kbytes of high-speed memory for the garbage collector data-structures and about 3% more main memory.

It is also practical to implement a memory subsystem incorporating these principles, with one or more processors sharing access to it.

The principles are also compatible with most modern programming languages. They would provide more efficient implementations along with checks for memory errors. Buffer overflows and dangling pointers are detected by hardware.

The architecture has the potential to save energy. Along with a reducing the operations needed for memory allocation, the garbage collector creates a single region of unused memory which can be powered down.

Memory allocation

Memory is allocated in *tuples*. A GETM instruction allocates a new tuple containing n words numbered from 0 to $n-1$, and produces a *pointer* to word 0 of the tuple. A pointer differs from a data value. Every word in memory contains information to specify whether it holds a pointer value or a data value. The processor registers also contain information to specify whether they hold a pointer value or a data value; registers such as a program counter pc or a stack pointer sp always hold pointer values.

Each instruction checks to ensure that the register contents are appropriate for the operation it performs. For example, an instruction to load a word from memory must have a pointer operand. The memory access instructions that use a pointer to a tuple along with an offset check that the specified location falls within the tuple.

When a tuple is no longer needed, the memory space it occupies is recovered automatically by a *garbage collector* implemented by hardware in the processor. The garbage collector runs continually; each cycle of the garbage collector marks all of the tuples in use, then copies them all towards the bottom of the memory, recovering space occupied by the tuples no longer in use. Newly created tuples occupy space above the existing tuples, at an address held in a register $heappoint$.

To support the memory allocation and garbage collection, a tuple $directory$ is used. Each tuple has a corresponding entry in the directory which contains the address of the tuple in memory and the number of words in the tuple. It also has space to hold the address of another tuple which is used to form a list of unused directory entries (and is also used by the garbage collector). When a memory access is performed using a pointer, the most significant bits of the pointer are used as a *handle* to address a directory entry. This provides the memory address of word 0 of the tuple; the least significant bits of the pointer determine the offset of the word to be accessed. A check is made to ensure that the offset is less than the size of the tuple.

Each tuple in memory has a control word which contains the handle of the directory entry corresponding to the tuple. It can also be used to hold a small value to indicate, for example, the type of the tuple. This is known as a tag.

The garbage collector is implemented as a state machine. Each state transition performs at most one memory access. The state transitions are normally performed when the memory is not required for instruction fetch or instruction execution. If the memory or the directory is full, instruction execution stops until the garbage collector completes its current cycle.

Addresses and the Directory

Each word w in memory or in a register holds a pointer value or a data value w_{value} and a flag w_{ptr} which is $true$ if w is a pointer, $false$ otherwise. If w is a pointer, then w_{value} has two components:

w_{handle}	identifies a directory entry of a tuple
w_{offset}	identifies an address within a tuple

The handle is represented by the upper half of the word, the offset by the lower half.

Each directory entry d of a tuple has four components:

d_{addr}	the address of the tuple in memory
d_{size}	the size of the tuple (in words)
d_{mark}	the marking flag for garbage collection
d_{deep}	a flag to indicate whether the tuple contains pointers

For a 32-bit wordlength, there will be up to 65536 tuples each of size up to 65536 bytes and the directory will have 65536 entries each with 48 bits (only 14 are needed for the length).

The Garbage Collector - Marking

The *marking* process uses two lists, $current$ and $next$. The $current$ list contains tuples that have been marked but which need to be scanned to determine if they contain pointers to other tuples that must be marked. Tuples are taken from the current list and their contents are examined; when a pointer is found, the corresponding tuple is marked and added to the $next$ list. When the $current$ list is empty, normally the $next$ list wlll contain new tuples to be scanned; the $next$ list replaces the $current$ list and scanning continues. If the $next$ list is empty when the $current$ list becomes empty, the marking is complete and the garbage collector will move to the *sweeping* process.

During the marking process, it is possible that new tuples will be created; these are added to the $next$ list. This also occurs whenever a tuple is assigned to a location in another tuple.

It is common for there to be tuples which contain data but no pointers. Each directory entry has an additional $deep$ marker. This is cleared when a tuple is created and is only set when a pointer is assigned to a location in the tuple. It is used to prevent unnecessary scanning during the marking process.

As the marking proceeds, a record $livesize$ is kept of the total amount of space that will be used by tuples that have been marked and that will be retained; this is used to optimise the sweeping process.

The Garbage Collector - Sweeping

The *sweeping* process accesses all of the locations in memory up to the $heappoint$. It uses two address registers, src and $dest$ to copy each tuple from its current location to its final location. It uses the control word of each tuple to access the tuple's handle. This enables it to determine the size of the tuple and whether the tuple is marked. If the tuple is marked it is copied one word at a time from src to $dest$. This copying step is omitted if src and $dest$ are equal; this occurs if all tuples up to the current value of src have been retained.

The marking process can only operate correctly if newly created tuples are initialised so that their components are data (not pointer) values. This means that when marked tuples are moved, the locations they move from must be re-initialised. However, there is no need to do this if a tuple (or a tuple component) is moved from a location that will be occupied by other tuples moved during the sweeping process. This is controlled by comparing the source addresses against $livesize$. Similary, the locations occupied by unmarked tuples must be re-initialised unless they will be occupied by tuples moved during the sweeping process.

The sweeping process terminates when the src address matches the $heappoint$. Additional tuples may be created during the sweeping process; these will be marked as they are created so that they will be copied. When the sweeping process terminates, the $heappoint$ is set to the $dest$ address.

Memory allocation and access

Memory for a tuple is allocated using one of the GETM instructions. This allocates a handle from the list of free handles and also allocates memory for the tuple at the $heappoint$. The control word of the new tuple is set and the tuple is marked.

The garbage collector operates at the same time that the processor is performing

memory accesses. It is therefore possible that a tuple is being moved during the sweeping process when the processor attempts to read or write locations within it. This is detected by the hardware and the read or write access is made to the actual location of the data.

When writing a pointer to memory during the marking process, the tuple addressed by the pointer must be marked and added to the $next$ list. Otherwise it would be possible for a tuple to remain unmarked because the pointer to it is moved by the executing program from a tuple that has not yet been marked into one that has already been marked. Also, if a pointer is written to a location in a tuple, the tuple must be marked by setting its $deep$ indicator to ensure that its contents are scanned during the marking process.

An experimental processor

The memory allocation system described above has been tested using a simple processor in conjunction with a simple programming language and self-hosting compiler. The initial results of this suggest that in many practical situations, the garbage collector will be able to retrieve unused memory space as fast as new space is allocated.