A Processor with List Structured Memory

P.A. Rounce

Department of Computer Science, University College London
Gower Street, London, WC1E 6BT, England

1. Abstract

This paper discusses the Sprint processor and its list structured memory. It describes the memory structure and its special features. It then examines how this style of memory has been supported on standard memories and the implementation of the Sprint processor.

2. Introduction

Sprint is a processor developed within ESPRIT PROJECT 1588, the SPAN project[3]. Sprint is designed as the computing element for a parallel processor, and has an associated communications device for inter-processor communication for a message-switching network, although other communications styles can be used[4]. Sprint has been designed and manufactured to implement efficiently the Kernel System of the SPAN project[1,2], and it was this model architecture, that primarily directed the design. Secondary considerations were to provide a generally applicable system with support for operating systems etc, and to have efficient operation, measured in instruction rate, instruction density and ease of use.

The major feature of Sprint (and of the Kernel System) architecture is that it has a totally list-structured memory, holding both code and data. Each processor has its own local, but not private, list-structured memory. Inter-processor communications is point-to-point and is performed through operations on these local memories, which form the global memory of the whole system. Local and non-local memory are addressed in identical fashion with identical operations, and the difference is recognised by the address used: a global addressing scheme. Operations on memory include both shared memory accesses, i.e. load and store, and message passing accesses, i.e. put and get. To support the latter, memory has an empty state, such that a put can occur on a memory location in this state, but not a get, and vice-versa.

3. List-Structured Memory

A list is a sequence of memory locations, which can be randomly accessed via a selector: the position of the location in the list. In the Kernel System the length of a list is unbounded except by the limit on the magnitude of an integer (32 bits) (the length can be zero), and lists are primitive objects like integers. A memory location can be empty, or it can hold an integer or it can hold a list. If a location holds a list, this list can hold other lists, and so on. Memory locations only exist within lists, and the whole of a local memory is held in a single list, Fig 1. Memory is created and deleted, as lists are created and deleted: the memory structure is dynamic.

Why a List-Structured Memory?

The architecture is designed to support both numeric and symbolic computing. The list-structured memory is designed to provide support for both these. For symbolic, having the list structured memory with lists and list operators as primitive objects, and with a machine designed to operate these efficiently, is expected to improve the mapping of symbolic programmes on to the machine and improve execution time: simulations on a standard workstation give some support to this, although these tests are limited. For numeric computing, the use of list-structured memory is a bonus: arrays and structures naturally map on to lists, arrays of structures map on to lists of lists, the hardware bounds checking will trap erroneous references. Of course, for numeric computing, random access is necessary for any sort of efficient operation and for code access: why not for symbolic?

4. Mapping a Program on to the Memory

Figure 2 shows how a C-program might be mapped on to memory as a single list with sub-lists. All globally-named objects, i.e. global data and functions, are stored in the top-level program list for ease of addressing: a name maps to a selector into this list. Data arrays naturally map into lists, but do so functions and data structures, and the bounds checking hardware can trap out of bounds excursions on any of these. The stack is just a list stored in a convenient place, and the heap can be implemented as a list of lists with each new allocation creating a new list of the appropriate length and struc-
ture. If all addressing is performed relative to the top-level program list, as is possible with Sprint, then the list is a totally relocatable object, which can be loaded into any memory element and also can be moved at any subsequent stage of execution, even migrated to another processor. A list-structured memory thus seems a very natural way to map code, data and programs on to memory.

**On treating Lists as Primitive Objects**

Most conventional architectures treat just integers and addresses as primitive objects, although many now do the same for floating point data. By this I mean that the architectures provide hardware support for them, e.g. registers for holding them, instructions for moving, copying and operating on them, addressing modes or instructions for transferring them to and from memory. Sprint attempts to provide the same form of support for lists. The Kernel System's model architecture provides just this. Lists can be read and written with a single instruction; both memory and registers can hold any size and structure of list in a single location; instructions are provided for creating, truncating, extending and joining lists; memory operators operate on lists as well as integers. Of course, it's all very easy in a model architecture. For the message-passing operators, this gives rise to no difference between the treatment required for integers or lists. The get operation pulls the object from memory and marks the memory location as empty, while put stores an object only into a location marked empty. The application of the shared memory operators to lists leads to some complication. The load operation requires a copy of an object to be brought in from memory, while a store overwrites some object in memory, which is deleted (the get and put operators have no such requirements). Thus load implies a list copy if a list is accessed, while store implies list deletion if a list is overwritten.

For the implemented hardware, treating lists as primitive objects is much more difficult, since lists are complex structures. Many of the operations to achieve this have been implemented in hardware in the processor, but a number of operations have currently to be implemented in software, although it has been shown that these could be performed with co-processor support. The key to the hardware support for lists is having a list pointer associated with each list, and manipulating this for much of the time instead of the list. The ability of the registers and memory to 'hold' a list is than reduced to a capability to hold a list pointer, with the list itself residing elsewhere. The message-passing operators can be directly applied, e.g. the get operator brings the list pointer into the processor, making empty the addressed memory location - a 'list' has been brought from memory. It is more complex for the shared memory operators with their implicit copy and deletion semantics, since the architecture is not designed to support multiple copies in memory of a list's pointer. The deletion on store is provided through a hardware exception to garbage collection software, when and only when a list is overwritten. For the copy implicit with load, two mechanisms are provided: Copy-on-Load through an exception when a load brings in a list, Copy-on-Store through marking the list pointer on load, and causing an exception, only if an attempt is made later to write it to memory. The latter is more efficient, since often a list is read just to find its length or do some other processing, and then the list is discarded: with Copy-on-Store this causes an unnecessary list copy, and a further unnecessary list deletion.

The use of a list pointer has also provided means to support bounds checking on list accesses by maintaining the length of the list in the pointer for ready access.

The processor does not have hardware to implement the more complex operations of list creation, extension, truncation, deletion etc., and these have to be done in software as yet. Direct hardware support is provided to traverse the list structure memory, and any number of levels in the memory can be traversed in a single operation, but taking multiple cycles to complete.

A common operation on a list is to extend or truncate it in place. With the Copy-Semantics of the Kernel System, using load-operate-store is inefficient, since there will be first a copy and then a deletion of the original, unless optimisation can be performed. Using get-operate-put is much more efficient for these 'in-situ' operations. This use could cause confusion with the intended purpose of these message-passing operators for semaphores, and Sprint provides support for 'fetch-operate-replace' operations, where the list is removed from memory and the memory is marked to be in use, but not empty, until it is returned.

**Addresses**

Addresses are themselves lists: adding a logical completeness to the system. The first element in an address list indicates a reference list in the memory and the remaining elements are selectors from this reference which specify the position of the addressed location. Figure 1 has an example of this, [root 4 4 3 2], which is an address originating from the top level list of the local memory. This form is an absolute address. Any lower level list can be used as a reference list, yielding relative addressing schemes. A global addressing scheme is built on the absolute form through use of a selector of 0: the first element of a list has selector 1. Thus [root 0 8 4 5] indicates to leave the local processor memory, go to processor 8 and then to apply selectors 4 and 5 to its root list. Only absolute addressing is available for global (non-local) addressing. Thus, a single addressing scheme is provided for both local and non-local addressing. The same memory operators can be applied, and the same code can be used without regard to the destination. This is very important for pointer usage. Sprint implements this uniform scheme by causing a hardware exception when the selector 0 is applied to the root list. The access can then be turned into a non-local access over the communications network via software or co-processor action. In fact, it is not the selector 0 that causes the exception, but the value returned from memory when the selector is applied, which mechanism may be useful at other times.

**4. The Hardware**

The processor details will only be briefly reviewed, although some implementation details are given, while other can be found in[4, 5]. Many of the actions of the processor require the identification of the data-type of an object - integer, list pointer, empty - and so a tagged architecture has been developed. This choice has provided many other advantages - type-checking of operands, address checking, improved instruction operation, increased code density. It is through the tagged architecture that implementation of many of the
mechanisms so far described has been possible. The processor is a RISC-style machine with only 32 instructions and 32 register references. A small 4-deep evaluation stack is provided in the register set. There are 3 modes of operation: Executive deals with exception handling and task scheduling, while Supervisor and User modes are provided for running tasks. These two modes are provided so that an operating system can be separated from User tasks. The processor mode is taken out of the chip, so that external memory protection can be applied.

A memory width of 40-bits has been provided, larger would have been used, but for problems in chip area (and expense) and pin-out. The 40-bit element width is controlled by the fields within the list pointer (fig 3) and not by the 3-bits of tag.

<table>
<thead>
<tr>
<th>Copy(1)</th>
<th>Tag(3)</th>
<th>Length(15)</th>
<th>Page Addr.(16)</th>
<th>Offset(5)</th>
</tr>
</thead>
</table>

**Figure 3. Structure of List Pointer**

**Memory Implementation**

To implement the list-structured memory, a novel algorithm has been developed, which provides true random access into lists: all locations in the same list can be accessed in equal time. It is a page-based scheme developed from knowledge of the system used in traditional page-based memory management. A paged scheme was chosen for the same reason as in memory management: the ease of free space management in a system where the memory allocation is dynamic. It is a tree-structured scheme. It supports list extension at head and tail without the requirement to copy the list fully. Each list requires its own instance of the tree structure: its memory elements are the leaves of the tree, grouped into pages, the branch nodes in the tree are page pointers down the tree structure, also grouped into pages, Fig 4.

**Figure 4. Mapping of a list into physical memory pages**

The number of levels between the top of the tree and the leaves depend on the maximum list size and the page size, fixed at 32K and 32 elements respectively. The tree depth varies with the list size and may be 1, 2 or 3 levels deep: Fig 4 shows a maximum depth structure. The tree structure is referenced by the list pointer, which contains the page address of the top level page, the list length, and the tree depth (within the tag), Fig 3. The offset field in the list pointer allows the list elements at the bottom level to be offset from the start of the 1st page at this level, allowing some free space at the head of the list to be used for list extension at the list head. List head extension never requires the elements of the list to be moved; if more space is needed at the list head an extra page is added and the list pointers at the higher levels are moved. This greatly reduces the expense of head extension; tail extension is much easier since most free space is at the end of the list. Application of a selector to a list pointer is a matter of adding the offset to the selector to produce a subscript (15-bits maximum), which locates the element in the bottom level of the tree structure. To traverse the structure the subscript is divided into 5-bit fields, which are each applied to a page at successive levels of the structure to locate the addressed element, Fig 4.

**Tags**

There is 3-bit tag field in each datatype, giving 8 datatypes grouped as 4 'list' datatypes and 4 non-'list' ('atomic') datatypes. The latter are integer, empty, specialatom and 'context/non-local'. The last is used for 2 purposes: one to mark the first element in an address list, the context, so that address lists can be validated on application, and two to generate a non-local exception when an instance of the datatype is read in during address processing. The 'Special-atom' datatype is available for general use, and there is support for it to be used in 'fetch-operate-replace' operations. 3 of the list datatypes mark varieties of the list pointer and identify the number of levels in the underlying list's structure (1, 2 or 3), while the 4th is for general use as a datatype, although categorised within the 'list' class. Besides requiring extra bits in both memory, registers and data busses, the provision of tags required extra instructions to insert/extract tags from objects, extra codes for conditional branching/execution, and extra control logic to provide tag checking. There are 32 conditional branch tests of which 11 are tag tests on a data object, e.g.:-

```
test&branch holds list?, register
```

Having tagged data objects affects many parts of the control logic, but only 3 instances will be mentioned. One logic block checks the tags on incoming data objects from memory to implement get and put instructions, Copy-on-Store exceptions, etc. Another checks tags on incoming instructions so that non-integer code objects are not executed, but are moved to the evaluation stack (this simplifies the loading of non-integer data); it also deals with conditional code execution, through which there is a capability to bring in 32-bit integers directly to the stack. A third block checks the tags on operands on the datapath during instruction execution. This requires the tags to be latched during operand transfer on the datapath and brought out to the control logic, where a test code from the instruction decoder signals the valid datatypes for the instruction. An exception is generated if the datatypes are invalid.

**Memory Accessing**

For reasons of consistency in the dynamic environment of the memory structure, physical addresses, other than in list pointers, are rarely held. They are found, used and discarded; multiple copies of list pointers are not expected to be stored in the memory. Physical addresses should be calculated as needed from the logical (list) address. There are 2 instruction forms for doing this calculation. One, translate, applies a single selector to a list pointer; this is very fast (2, 4, 5 instruction cycles for 1, 2 and 3-level list structures), and is available for minimal (2 element) list
addresses known at compile time. The other form, evaluate, takes a list pointer to a list address (with any number of elements) and traverses the list structure to locate the required memory location. Both these forms are done in indivisible sequences in hardware. These forms can be coupled with any of the memory operation instructions, load, store, put, etc., to perform the required operation. It has been possible, because of the small number of instructions and registers to get a single instruction into 16-bits. Thus, it is possible in many cases to get 2 instructions into a memory element, along with an integer tag and a 4-bit conditional execution code. The instruction-pair, so created, are executed as an indivisible pair by the processor. It is expected that an address calculation and memory operation will be executed as such an instruction-pair, so that a physical addresses will be produced and consumed within the pair, and there is no possibility for the sequence to be accidentally separated by interrupt or bus arbitration. Such a sequence may be broken by a hardware exception, e.g. by the Copy-on-Store exception.

The following code sequence of code for a put to memory illustrates a number of Sprint features:

```
[root 1 2 3 4]
evaluate stack1 ; put stack2
?if_not-empty? system call ; subtract #2, ip
```

Each line provides one 4-bit word of code. The first word is a list pointer to a list holding the list address [root 1 2 3 4]. On fetching the list pointer, it is placed on the evaluation stack. The evaluate instruction processes the list address, traversing the memory to locate and read the addressed location. The data object returned by this read controls whether the put instruction writes a value from the 2nd location on the stack to memory: only if the read value has the datatype empty. The 3rd word is only executed if the value returned by evaluate is not empty, in which case a system call is made to the Executive mode to block the current process. When the process is later resumed the subtract is executed, which decrements the code selector register, ip, so that the evaluate and put instructions are retried.

**Instruction Fetch**

Code is stored in and fetched from lists. This requires that the processor holds the list pointer to the current code list and the selector to the next element to execute in this list. Doing address calculations on these values on each fetch would be a large overhead, 2-5 clock instruction cycles per fetch, and this is reduced by also holding the physical address of the next location to be accessed. This is incremented on each fetch and only on page overflow or on a branch does the full list calculation have to be done to provide the new physical address. Instruction fetch is pipelined with instruction execution whenever possible. This is the only pipelined activity.

**Memory Cycles**

All memory cycles complete within an instruction cycle, and there is a capability to extend a cycle with extra clock periods. To reduce recourse to this, all memory cycles require the address of the access to be loaded to the output data address register during the previous cycle. The address can then be output very early during the memory access cycle, along with direction signals, so that device selection occurs early enough for access times to be sufficient for minimum length cycles in most cases. To support this there are 2 output registers, one for data and the other holding the physical address for the next code fetch. Both these have integral increment units, and the data register is accessible as a standard register for reading and writing.

**Datapath**

A fairly standard, precharged, 2-bus datapath has been developed, containing many familiar elements - registers, alu with 4-bit carry lookahead, 32-bit rotate unit. The evaluation stack is developed from 4 standard registers. Which register is to be accessed during a stack reference is selected by control logic, using top of stack and stack depth information held in the processor status register (effectively a finite state machine). To support bounds checking on list accesses, a number of comparators are present. In particular, there is a special register unit containing a 40-bit list pointer register, a 15-bit selector register with an integral increment unit and a 15-bit comparator. One instance of this unit is used to hold the code list pointer and code list selector, so that the selector can be incremented after instruction fetch and the new value checked against the current list length from the list pointer without requiring access to the datapath. The unit generates an exception signal if an out-of-bound access is attempted. There is also a bit field manipulation unit to insert and extract tag and length fields from objects plus a scan-path unit for checking the datapath.

**Control Logic**

At the heart of the control logic is the next-cycle selection logic and the instruction, source register, destination register and stack-reference decoders. The use of the decoders is maximised so that they operate during every processor cycle even if only to execute a NOP instruction during a non-pipelined instruction fetch or during system bus release. There are 3 internal sources of instructions to the decoders beside the external memory. There is a PLA which acts as a ROM and sources single instructions for the performance of special operations on the datapath, e.g. code register initialisation on processor reset, moving non-integer instruction words to the stack, moving exception vectors into the executive mode selector register. The other 2 sources are finite state machines which each provide a sequence of instructions to perform the multi-cycle translate and evaluate instructions: thus, these instructions use the single-cycle instruction set as a micro-code. The use of finite-state machines in this way improves performance over placing this code in external software: the code density is increased, indivisible execution is provided, testing and branching can be performed in parallel by the hardware, instruction fetch overhead is removed. Other blocks contain logic for exception handling, instruction pre-fetch (pipelined), memory access control, clock generation and scan-path control.

**Processor Timing**

The processor is driven by 2-phase non-overlapping clocks. An instruction cycle is 2 clocks in length.
and is divided into 4 quarters, Q1-Q4, of a half-clock period each. In Q1 and Q3 the datapath is precharged, while in Q2 source operands are moved to functional units. Functional units operate continuously, but because of the use of precharge, they must complete processing of new operands by the end of Q3, so that a result can be returned in Q4. During Q4 of a cycle, the next-cycle logic determines what will be performed next: system bus release, process an exception, perform code pointer-selector translation, do a non-pipelined code fetch, execute instruction 1 or 2 of a pair. By the end of Q4, its outputs will have selected the source of the next instruction, and instruction decoding will have started. By the end of Q1, this will have completed, and control signals will be active to the datapath and other components. These are latched at this point and held until the next Q1 start. During Q2-Q4, these signals are conditioned by clock signals to control activities on the datapath. Conditional branch test results must be available by the end of Phase 3 for use by the next-cycle logic in Q4.

The Chip

The chip has been manufactured in 2 micron CMOS. Its overall size including pads is 71 mm² of which the datapath is ~21 mm², the control logic is ~15 mm² and the pads and routing make up the remaining ~35 mm². Scan-path test logic has been included that provides access to all the finite-state machine state bits, the datapath, the instruction bus, and all the signals between the control logic and datapath. The chip has been designed to run with a 10 MHz clock to give a 200 ns instruction cycle time.

5. Summary

A complete and consistent, random access, von Neuman, list-structured architecture has been presented, which offers many advantages over linear memory, when both numeric and symbolic processing are to be performed, and has some advantages even where just numeric is desired. There are many interesting and challenging issues in the implementation of a system around this architecture. A processor has been designed and manufactured, which supports it. It has an efficient random access mapping scheme, with hardware to perform the basic memory operations on it, and to signal automatically those conditions requiring more intensive operations. Its production demonstrates the feasibility of such a system.

Acknowledgement

The SPAN project was supported by ESPRIT as PROJECT 1588.

References.


