The Synthesis Kernel

John Ioannidis, Columbia University
Cestion P. Huang, MassaH and

abstract:

The Synthesis Kernel
1. Introduction

A trade-off between powerful features and efficient implementation exists in many operating systems. Systems with high-level interfaces and powerful features, like Argus\(^8\) and Eden,\(^7\) require a lot of code for their implementation, and this added overhead makes the systems slow. Systems with simple kernel calls, like the V kernel,\(^8\) Amoeba,\(^13\) and Mach,\(^1\) have little overhead and run fast. However, the application software then becomes more complex and slower because extra code is required to make up for the missing kernel functions. Our goal in developing the Synthesis distributed operating system is to escape from this trade-off. We want to provide a simple but high-level operating system interface to ease application development and at the same time offer fast execution.

To achieve our goal, we combine two concepts in Synthesis. The most important idea is the inclusion of a code synthesizer in the kernel. The code synthesizer provides efficiency through the generation of specialized code for frequently-executed kernel calls. For instance, when the programmer asks the operating system to open a file, special routines to read and write that specific file are returned. Through the generation of these frequently-executed system calls, Synthesis reduces operating system overhead. For example, typical Synthesis read routines average an execution path of 20 to 30 machine instructions. In contrast, the 4.3 BSD read call contains on the order of 500 lines of C code (Appendix B).

The second idea is an orthogonal interface called a synthetic machine. To a programmer, a synthetic machine presents a logical multi-processor with its own protected address space. Two reasons motivated this model of computation: to take advantage of general-purpose shared-memory multiprocessors, and to support the growing number of concurrent programs. The synthetic machine consists of three basic components: synthetic CPUs to run the program, synthetic memory to store the program and data, and synthetic I/O units to move data in and out of the synthetic machine.

The synthetic machine interface and kernel code synthesizer are independent ideas that have a synergistic effect. Without the code synthesizer, even a sophisticated implementation of synthetic machines would be very inefficient. Each high-level kernel call would require a large amount of code with a long execution time. Instead, the kernel code synthesizer generates specialized routines to make kernel calls short. Without a high-level interface, more layers of software would be needed to provide adequate functionality. The synthetic machine supplies high-level system calls to reduce the number of layers and the associated overhead. In Synthesis, application programmers will enjoy a high-level system interface with the efficiency of a “lean and mean” kernel.

To test these ideas, we have designed and implemented a prototype system, with a simplified kernel code synthesizer that implements a subset of the synthetic machine interface. Encouraged by positive results of the prototype, which confirmed our expectations on its performance, we are now implementing the full version of Synthesis.

In Section 2, we describe how the code synthesizer generates and optimizes code in the kernel. In Section 3, we summarize the synthetic machine interface and illustrate the power of the interface with an emulation of the UNIX system using synthetic machines. We outline the current Synthesis prototype in Section 4, including some measurements to illustrate the efficiency gained with synthesized code. Section 5 compares Synthesis with related work, and Section 6 concludes with a summary of progress.

2. Kernel Code Synthesizer

Typical operating system kernel routines maintain the system state in data structures such as linked lists. To perform its function, a kernel routine finds out the system state by traversing the appropriate data structures and then takes the corresponding action. In current operating systems, there are few short cuts to reach frequently-visited system states, which may require lengthy data structure traversals.

This research is partially supported by the New York State Center for Advanced Technology, Computer and Information Systems, NYSTC CAT(87)-3.
The fundamental ideas of kernel code synthesis is to capture functional invariants while ensuring that the code follows certain patterns. This process involves a combination of static analysis and optimization techniques to ensure that the generated code is both efficient and safe. Functional invariants play a crucial role in this process, as they help in ensuring that the code behaves as expected under various conditions.

In this section, we will discuss the concept of functional invariants and how they can be used to optimize kernel code. We will also explore some of the challenges associated with functional invariants and how they can be addressed.

2.1 Functional Invariants

Exclusive-join invariants

Exclusive-join invariants are a type of functional invariant that are useful in optimizing code. These invariants are used to ensure that certain operations are performed in a specific order, which can help in improving the performance of the code. Exclusive-join invariants are particularly important in the context of functional programming, where the order of operations can significantly affect the performance of the code.

In this section, we will discuss the concept of exclusive-join invariants and how they can be used to optimize code. We will also explore some of the challenges associated with exclusive-join invariants and how they can be addressed.
resulting code typically exhibits opportunities for further optimization, such as Factoring Invariants and elimination of data copying.

By induction, $F_{create}$ can eliminate the procedure call to the Session layer, and down through all layers. When we execute $F_{create}$, to establish a virtual circuit, the $F_{apply}$ code used thereafter to send and receive messages may consist of only sequential code.
The performance gain analysis is similar to the one in section 2.1.

2.3 Executable Data Structures

The executable data structures method is based on the observation that some data structures are traversed in some preferred order. Therefore, adding the executable code to the data structures to make them self-traversing may decrease the traversal overhead.

Let us consider the simplified example of the active job queue managed by a round-robin scheduler. Each element in the queue contains two short sequences of code: stopjob and startjob. The stopjob saves the registers and branches into the next job's startjob routine (in the next element in queue). The startjob restores the registers, installs the address of its own stopjob in the timer interrupt vector table, and resumes processing.

An interrupt causing a context switch will trigger the current program's stopjob, which saves the current state and branches directly into the next job's startjob. Note that the scheduler has been taken out of the loop. It is the queue itself that does the context switch, with a critical path on the order of ten machine instructions. The scheduler intervenes only to insert and delete elements from the queue.

2.4 Kernel Programmability

In the previous sections, we have described several innovative programming techniques to gain performance through code synthesis. Historically, the lambda functions in LISP have created S-expressions, which are executable in the LISP environment. However, new problems arose in our use of code synthesis on bare machine. Here, we summarize briefly the problems we have encountered and the approaches we have taken to solve them:

- Inflated kernel size due to code redundancy.
- Structuring of kernel and correctness of its algorithms.
- Protection of synthesized code.

One important concern in Synthesis is kernel size inflation due to the potential redundancy in the many $F_{small}$ and $F_{flat}$ programs generated by the same $F_{create}$. To solve this problem, $F_{create}$ generates either in-line code or subroutine calls using a mechanism similar to threaded code. Frequently invoked functions are expanded in-line into the user's synthetic machine and executed there. Rarely executed functions are stored in a common area, shared by all synthetic machines running threaded code. The decision of when to expand in-line is made by the programmer writing $F_{create}$. Although the size of the code synthesizer depends heavily on the kinds of facilities supported by the kernel, we hope to minimize the cost paid for code specialization. For example, the Synthesis $F_{create}$ calls correspond to the UNIX open system calls, with similar data structures and algorithms. The main difference between them resides in the actions taken by the system calls; open fills in the UNIX data structures, and $F_{create}$ places machine op-codes into an array. Therefore, we expect the cost of $F_{create}$ to be comparable to that of the UNIX open.

The structure of Synthesis kernel is superficially similar to a traditional operating system. Kernel calls of the $F_{create}$ type are invoked just like a normal system call. However, synthesized kernel calls created by $F_{create}$ are invoked in the kernel mode (usually through trap) through a branch into the synthesized code. By construction, the synthesized kernel calls perform a subset of the actions of normal kernel calls. These subsets calculate the same results and cause the same side effects for each specific case.

Synthesized code is protected through memory management. Each address space has its own page table, and synthesized code is placed in protected pages, inaccessible to the user program. To prevent the user program from tricking the kernel into executing code outside the protected pages, the synthesized routines are accessed via a jump table in the protected area of the address space. Since the user program can only specify an index into this table, the synthesized routines are entered at the proper entry
3.2 Synthetic Machine Influence

The synthetic machine influence that now we describe is that the synthetic machine is a model of the computer. It has been designed to resemble the computer closely. Although the synthetic machine is somewhat abstract, it is based on actual computer systems, and its instructions are essentially the same as those used in real computers. The synthetic machine has been designed to provide a clear understanding of the basic principles of computer operation. It is hoped that this description of the synthetic machine will be of value to students who are interested in the field of computer science.

3.3 Synthetic Machines

Synthetic machines are those that are designed to provide a clear understanding of the basic principles of computer operation. They are based on actual computer systems, and their instructions are essentially the same as those used in real computers. The synthetic machines have been designed to provide a clear understanding of the basic principles of computer operation. It is hoped that this description of the synthetic machine will be of value to students who are interested in the field of computer science.

3.4 Model of Computation

The synthetic machine is the model of computation. It is hoped that this description of the synthetic machine will be of value to students who are interested in the field of computer science.

The synthetic machine is the model of computation. It is hoped that this description of the synthetic machine will be of value to students who are interested in the field of computer science.
The synthetic machine interface is object-oriented. Synthetic machines and synthetic components are encapsulated resources, and users must ask them to perform the kernel calls defined in Table 1. Pioneer object-oriented systems such as Hydra,1 Eden,2 and Argus10 have achieved performance adequate for a prototype, while Mach is comparable to BSD UNIX.3 We believe that synthesized code will take the performance of Synthesis one step ahead of current systems. In Section 4.2, we justify our expectations with preliminary measurements.

3.3 Resource Sharing and Protection

To support the shared-memory model of parallel processing, all SCPUs within a synthetic machine share the same address space and thus, SMEMs can be used for communication and sharing between them. Semaphores control the synchronization between SCPUs. Currently, we are developing a high-level language to support concurrent programming with synthetic machines.

Efficient sharing between synthetic machines requires more care, since we enforce protection across their boundaries. SMEMs may be shared between two cooperating synthetic machines. The original creator of the SMEM sends the appropriate routines (synthesized for sharing) to its peer synthetic machine through a protected SIO channel. The peer then uses the routines to map the SMEM into its own synthetic machine. The access routines given to the peer determine the access privileges.

To support the message-passing model of parallel and distributed processing. SIOs include network traffic. Sharing SIOs is similar to sharing SMEMs, in that the creator of the SIO, e.g. a port for inter-process communication, also passes the access routines to the peer. Since the access routines are passed through the protected SIO channel, no forging is possible. This protection mechanism is more flexible than that achieved by capabilities with a constant number of bits, since these routines can implement any kind of control, for example, access control lists. Furthermore, the creator may reconfigure or resynthesize the SMEM or SIO in such a way as to invalidate the earlier access routines, thus revoking access rights already conceded.

Table 1: Examples of Synthetic Machine Kernel Calls

<table>
<thead>
<tr>
<th>create (generator)</th>
<th>creates an SMACH</th>
<th>creates a thread of control</th>
<th>allocates memory</th>
<th>creates ports, opens files, allocates devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminate (executive)</td>
<td>kills an SMACH and all its components</td>
<td>kills a thread of control</td>
<td>frees memory</td>
<td>kills ports, closes files</td>
</tr>
<tr>
<td>reconfigure (executive)</td>
<td>resuming/suspending an SMACH, changing its priority, waiting on event</td>
<td>resuming/suspending SCPU, changing its priority, initiating sharing</td>
<td>changes protection, initiates sharing</td>
<td>lseek, changes protection</td>
</tr>
<tr>
<td>query (executive)</td>
<td>gets priority, gets SMACH id, gets uid</td>
<td>gets priority, gets state, gets SCPU id</td>
<td>gets size, starting address</td>
<td>gets device type, state, device pointer</td>
</tr>
<tr>
<td>read (executive)</td>
<td>unused</td>
<td>unused</td>
<td>unused</td>
<td>reads file, rec. messages, and any other input operation</td>
</tr>
<tr>
<td>write (executive)</td>
<td>unused</td>
<td>unused</td>
<td>unused</td>
<td>writes file, sends messages, and any other output operation</td>
</tr>
</tbody>
</table>

To illustrate the use of these calls, we give an SIO create example, which opens a file. Given the file name and a read/write option, it returns the code for the executive calls. You call read to read from the file and write to write to it. To find the file length, current seek position and other file attributes, you use query. The reconfigure call changes those attributes. Finally, terminate closes the file.
with the symbolic machine interface.

features of our architecture, and maintain rigorous compatibility
information with symbolic processors. Each processor
provides an interface to the symbolic processor. Each
processor is interfaced to the hardware and software
compatibility. Each processor includes an interface to the
symbolic processor. Each processor includes an interface to the
symbolic processor.

In the description of the symbolic processor, the symbolic processor
interfaced to the hardware and software compatibility. Each
processor includes an interface to the symbolic processor. Each
processor includes an interface to the symbolic processor.

The symbolic processor includes an interface to the hardware and
software compatibility. Each processor includes an interface to the
symbolic processor. Each processor includes an interface to the
symbolic processor.

In the description of the symbolic processor, the symbolic processor
interfaced to the hardware and software compatibility. Each
processor includes an interface to the symbolic processor. Each
processor includes an interface to the symbolic processor.

The symbolic processor includes an interface to the hardware and
software compatibility. Each processor includes an interface to the
symbolic processor. Each processor includes an interface to the
symbolic processor.

In the description of the symbolic processor, the symbolic processor
interfaced to the hardware and software compatibility. Each
processor includes an interface to the symbolic processor. Each
processor includes an interface to the symbolic processor.

The symbolic processor includes an interface to the hardware and
software compatibility. Each processor includes an interface to the
symbolic processor. Each processor includes an interface to the
symbolic processor.

In the description of the symbolic processor, the symbolic processor
interfaced to the hardware and software compatibility. Each
processor includes an interface to the symbolic processor. Each
processor includes an interface to the symbolic processor.

The symbolic processor includes an interface to the hardware and
software compatibility. Each processor includes an interface to the
symbolic processor. Each processor includes an interface to the
symbolic processor.

In the description of the symbolic processor, the symbolic processor
interfaced to the hardware and software compatibility. Each
processor includes an interface to the symbolic processor. Each
processor includes an interface to the symbolic processor.

The symbolic processor includes an interface to the hardware and
software compatibility. Each processor includes an interface to the
symbolic processor. Each processor includes an interface to the
symbolic processor.

In the description of the symbolic processor, the symbolic processor
interfaced to the hardware and software compatibility. Each
processor includes an interface to the symbolic processor. Each
processor includes an interface to the symbolic processor.

The symbolic processor includes an interface to the hardware and
software compatibility. Each processor includes an interface to the
symbolic processor. Each processor includes an interface to the
symbolic processor.

In the description of the symbolic processor, the symbolic processor
interfaced to the hardware and software compatibility. Each
processor includes an interface to the symbolic processor. Each
processor includes an interface to the symbolic processor.

The symbolic processor includes an interface to the hardware and
software compatibility. Each processor includes an interface to the
symbolic processor. Each processor includes an interface to the
symbolic processor.

In the description of the symbolic processor, the symbolic processor
interfaced to the hardware and software compatibility. Each
processor includes an interface to the symbolic processor. Each
processor includes an interface to the symbolic processor.

The symbolic processor includes an interface to the hardware and
software compatibility. Each processor includes an interface to the
symbolic processor. Each processor includes an interface to the
symbolic processor.

In the description of the symbolic processor, the symbolic processor
interfaced to the hardware and software compatibility. Each
processor includes an interface to the symbolic processor. Each
processor includes an interface to the symbolic processor.

The symbolic processor includes an interface to the hardware and
software compatibility. Each processor includes an interface to the
symbolic processor. Each processor includes an interface to the
symbolic processor.

In the description of the symbolic processor, the symbolic processor
interfaced to the hardware and software compatibility. Each
processor includes an interface to the symbolic processor. Each
processor includes an interface to the symbolic processor.

The symbolic processor includes an interface to the hardware and
software compatibility. Each processor includes an interface to the
symbolic processor. Each processor includes an interface to the
symbolic processor.

In the description of the symbolic processor, the symbolic processor
interfaced to the hardware and software compatibility. Each
processor includes an interface to the symbolic processor. Each
processor includes an interface to the symbolic processor.

The symbolic processor includes an interface to the hardware and
software compatibility. Each processor includes an interface to the
symbolic processor. Each processor includes an interface to the
symbolic processor.

In the description of the symbolic processor, the symbolic processor
interfaced to the hardware and software compatibility. Each
processor includes an interface to the symbolic processor. Each
processor includes an interface to the symbolic processor.

The symbolic processor includes an interface to the hardware and
software compatibility. Each processor includes an interface to the
symbolic processor. Each processor includes an interface to the
symbolic processor.

In the description of the symbolic processor, the symbolic processor
interfaced to the hardware and software compatibility. Each
processor includes an interface to the symbolic processor. Each
processor includes an interface to the symbolic processor.

The symbolic processor includes an interface to the hardware and
software compatibility. Each processor includes an interface to the
symbolic processor. Each processor includes an interface to the
symbolic processor.

In the description of the symbolic processor, the symbolic processor
interfaced to the hardware and software compatibility. Each
processor includes an interface to the symbolic processor. Each
processor includes an interface to the symbolic processor.

The symbolic processor includes an interface to the hardware and
software compatibility. Each processor includes an interface to the
symbolic processor. Each processor includes an interface to the
symbolic processor.

In the description of the symbolic processor, the symbolic processor
interfaced to the hardware and software compatibility. Each
processor includes an interface to the symbolic processor. Each
processor includes an interface to the symbolic processor.

The symbolic processor includes an interface to the hardware and
software compatibility. Each processor includes an interface to the
symbolic processor. Each processor includes an interface to the
symbolic processor.

In the description of the symbolic processor, the symbolic processor
interfaced to the hardware and software compatibility. Each
processor includes an interface to the symbolic processor. Each
processor includes an interface to the symbolic processor.

The symbolic processor includes an interface to the hardware and
software compatibility. Each processor includes an interface to the
symbolic processor. Each processor includes an interface to the
symbolic processor.

In the description of the symbolic processor, the symbolic processor
interfaced to the hardware and software compatibility. Each
processor includes an interface to the symbolic processor. Each
processor includes an interface to the symbolic processor.

The symbolic processor includes an interface to the hardware and
software compatibility. Each processor includes an interface to the
symbolic processor. Each processor includes an interface to the
symbolic processor.

In the description of the symbolic processor, the symbolic processor
interfaced to the hardware and software compatibility. Each
processor includes an interface to the symbolic processor. Each
processor includes an interface to the symbolic processor.

The symbolic processor includes an interface to the hardware and
software compatibility. Each processor includes an interface to the
symbolic processor. Each processor includes an interface to the
symbolic processor.

In the description of the symbolic processor, the symbolic processor
interfaced to the hardware and software compatibility. Each
processor includes an interface to the symbolic processor. Each
processor includes an interface to the symbolic processor.

The symbolic processor includes an interface to the hardware and
software compatibility. Each processor includes an interface to the
symbolic processor. Each processor includes an interface to the
symbolic processor.

In the description of the symbolic processor, the symbolic processor
interfaced to the hardware and software compatibility. Each
processor includes an interface to the symbolic processor. Each
processor includes an interface to the symbolic processor.

The symbolic processor includes an interface to the hardware and
software compatibility. Each processor includes an interface to the
symbolic processor. Each processor includes an interface to the
symbolic processor.
code synthesizer

from both by the high-level synthetic machine instruction and the
software instruction. In this way, both the RISC and the SYNIX
systems have been made to operate in parallel.

The V Kernel and another are two examples of small

5. Comparison with Other Systems

code is non-trivial, and we need heuristics to support it.

applications, and the programs are in the high-level
language. Thus, a high-level language is the key to the

Table 2: Measured Sizes and Comparison

Masscomp

The results appear in the same table, columns HP, SUN-3, and

McSOAC model 5020, with the RTX UNIX 3.1 operating system.

HP-UX 1.0, SUN-3/20 with the SUN-OS 3.2, and Masscomp.

same program were run on the HP 9000/3270 with the

The results from the second stage are shown in

Masscomp front-end synthesizer. The second stage

6080-based machine.

The numbers actually measured on the prototype system

overhead of a system call is measured over a larger amount of

In this example, we have a more common situation where the
Mach\(^1\) offers an object-oriented interface that is isomorphic to a specialized synthetic machine. A Mach task corresponds to a synthetic machine; Mach thread, an SCPU; Mach port, the network SIO; Mach messages, network SIO read and write; and Mach virtual memory, an SMem. The synthetic machine uses the same interface for all I/O activities, and child synthetic machines may be nested within a parent. As with other systems, Mach does not use a kernel code synthesizer.

Emerald\(^2\) is an object-oriented, integrated language and system. Synthesis lacks the language support in Emerald, in particular the sophisticated typing system. In compensation, although Emerald objects may be constructed at run-time (in a way similar to synthesized code), its kernel calls are not synthesized.

6. Conclusion

We have combined two ideas in Synthesis. First, a kernel code synthesizer produces specialized and extremely efficient code for system calls. Second, an orthogonal, object-oriented, and high-level interface has been derived from a simple model of computation. This combination gives Synthesis unique advantages. The kernel code synthesizer reduces the high-level interface inefficiency problem. The high-level interface removes the slowdown due to multiple layers of software built on small kernels.

Efficiency derived from the code synthesizer has been demonstrated on a prototype system. An example is a specialized read system call, which takes about fifteen microseconds. In comparison, the HP-UX and Masscomp RTU systems running on similar hardware need a few hundred microseconds for an equivalent, non-specialized read call. We expect to do even better in our full system, with very efficient SIO kernel calls including file systems, network communications, and other devices.

We are implementing the full version of Synthesis for SUN-3 workstations and a 68020-based machine. After the kernel, we will design and implement language support, transaction processing, and real-time support, all taking advantage of synthesized code.

We believe that the unique combination of simplicity and efficiency makes Synthesis an excellent system to write and execute programs.

Acknowledgments

We would like to thank Perry Metzger, Mike Schwartz, and Eric Jul for their comments that improved the presentation of the paper. Gail Kaiser, Chip Maguire, Rajendra Raj, and Jonathan Smith helped with a previous version of the paper. Equally important, we would like to thank Ed Hee and Paul Kanevsky for their varied contributions to the project, and Al Lash for assistance in procuring laboratory facilities. AMD, Hitachi, Intel, Motorola, and Mupac contributed with hardware parts for the project.
we summarize its actions in Appendix B. As the result of applying all these optimizations. The execution of
the full optimizer is implemented in instructions (2) and (4).

When we describe in Figure 2 the optimized code symbols we
expect to produce in the full version. We will now explain what is
the status word (15), and the system call exits (16).
the function exits (17). Finally the registers are restored (18),
and the OK status code is signaled (19), and the execution of
the DECnet CPU, The full version of the code produces the
number (3) and the stack position is added to the base memory
address (7). The stack is used (10), and the stack position is
added to the base memory for all addresses of the special registers read through the 1/0 device, for all addresses of the
kernel data space instructions (3) and (4) and all registers.

In Figure 1 we show the trace produced by the execution of

Figure 1: Trace of Code Actually Measured

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
</tbody>
</table>

Figure 2: Trace of Expected Code

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
<tr>
<td><code>move.W</code></td>
<td>0000000000</td>
</tr>
</tbody>
</table>

Appendix A
References
13. D. M. Ritchie and K. Thompson, The UNIX Time-sharing System, 

14. J. E. Stoy, Denotational Semantics: The Scott-Strachey Approach to 

15. A. S. Tanenbaum and S. J. Mullender, The Design of a Capability-
    Based Distributed Operating System, Technical Report IR-88, 
    Department of Mathematics and Computer Science, Vrije 
    Universiteit Amsterdam, November 1984.

16. W. A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, 
    and F. Pollack, Hydra: the kernel of a multiprocessor operating 

[submitted Sept. 22, 1987; accepted Nov. 5, 1987]