ON DYNAMICALLY UPDATING A COMPUTER PROGRAM: FROM CONCEPT TO PROTOTYPE

OPHIR FRIEDER and MARK E. SEGAL

Reprinted From: The Journal of Systems and Software
Volume 14 Number 2 February 1991

On Dynamically Updating a Computer Program: From Concept to Prototype

Ophir Frieder[1]
Department of Computer Science, George Mason University, Fairfax, Virginia

Mark E. Segal[2]
Bellcore, Morristown, New Jersey

An approach to dynamically updating a computer program, i.e., updating while it is executing, is presented. Dynamic updating is crucial in applications where the cost of stopping and restarting the program makes doing so impractical. The presented system works with programs written in procedural languages such as Pascal and C. It is assumed that computer programs are written in a top-down manner consistent with good software engineering practices. Also assumed is that the underlying computer system logically provides a network-wide sparse virtual address space. Using these assumptions, it is possible to update computer programs with minimum interruption to the running program. By partitioning the address space into a number of version spaces, the handling of multiple simultaneous updates is possible. This allows one update to begin before previous updates complete. Via appropriate mapping mechanisms, old versions of procedures may call new procedures and maintain consistency. An overview of the design and implementation of a working prototype updating system is discussed and a sample updating session is illustrated.

1. INTRODUCTION

By its very nature, computer software is constantly changing. Change may be necessary because new features were added to a program or because bugs were discovered in the current version. Sometimes change may be necessary when the specifications of the tasks a program must perform, how it must perform them, or the environment where the tasks must be performed have changed. Once the appropriate modifications have been made to a program, the old version may be stopped and the new version may be loaded and run. There are circumstances, however, where temporarily stopping a program while a new version is being loaded is not viable. This is primarily due to the significant cost of such an operation. This cost might be manifest in lost revenue (an airline reservation system or a telecommunications switching system) [2, 5, 11] or in terms of danger to human life (a computer-controlled life-support system or an air-traffic control system). The ability to dynamically update a program, i.e., load a new version of a program without stopping the currently running version, could alleviate these costs in many cases.

An approach to dynamic program updating is described. The approach updates programs written in procedural programming languages by replacing the program's individual procedures. This approach can be used to update distributed programs [21] across a network of machines by performing a similar updating sequence to that done on a single machine. The main limiting assumption of the presented approach is the requirement that programs be written in procedural programming languages using a top-down design methodology. As the top-down design approach is consistent with good software engineering practices, no attempt is made to weaken this precondition.

The remainder of this paper is organized as follows: Section 2 reviews previous research in dynamic updating of computer programs. An overview of the updat-

Address correspondence to Mark E. Segal, Bellcore, 445 South Street, Room 2A275, Morristown, NJ 07962-1910.

[1]Portions of this work were performed while this author was at Bellcore, Morristown, NJ.

[2]Portions of this work were performed, while this author was with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI.

0164-1212/91/$3.50

ing system is presented in Section 3, with an in-depth examination of the updating system architecture comprising Section 4. In Section 5, a description of a prototype updating system is provided. An illustrative sample update is included in Section 6. Finally, a conclusion is provided in Section 7.

2. PREVIOUS WORK

We classify prior approaches to the problem of replacing portions of computer programs without stopping them into three main categories: hardware-based, service-oriented, and procedural. Hardware-based approaches attack the problem by providing a redundant CPU and peripherals to be configured with a new version of the program while the old one continues to run [19]. When the program is updated, the old system is physically disabled while the new one is enabled.

Software-based, service-oriented approaches attack the problem by imposing a server/client relationship on the programs they can update [4]. In such an approach, a number of clients request a service from a server via some well-defined mechanism such as an operating system primitive or a remote procedure call [3]. A server may be updated by temporarily disabling its services and then installing a new server. Although this approach executes in a distributed system, it will only work with software systems that observe a server/client relationship.

The Conic System [26] uses a variation of this idea. A Conic distributed program is divided into a set of modules that communicate with other modules via a set of software "links" between the modules. A Conic program is dynamically updated by changing the links from connecting old versions of modules to the new versions. Although Conic programs are not structured in a server/client fashion, each module still must be replaced as a single unit. As with service-oriented approaches, this replacement mechanism may not be appropriate for programs built from large modules since small changes within a module require the entire module to be replaced.

Finally, software-based, procedure-oriented approaches attack the problem by replacing individual procedures as the program executes. In such an approach when all of the "old" procedures have been replaced by all of the "new" procedures, the program has been updated. This class of updating system is related in some respects to dynamic-type replacement systems such as Fabry [13]. In a type replacement system, the routines providing access to abstract data types are replaced while the program using them continues to run. Although this type of system allows abstract data type implementation to be changed between versions of a program, it does not address the more general issues of code restructuring, such as interface changes. The DMERT [25], the Secure On-the-Fly Method [6], and the DAS operating systems [15] all provide mechanisms for replacing the individual procedures that comprise a program. These systems only address the case where the specification (parameters and return values) of the procedures being updated have not changed and are thus limited to those particular circumstances.

The DYMOS System [17] is a complete dynamic updating system. It provides editors, compilers, and a shell to facilitate updating a computer program written in the StarMod language [8]. DYMOS will work in a tightly-coupled multiprocessor but does not scale well to a distributed system since it requires a complicated locking protocol for every procedure invocation regardless of whether or not an update is actually being performed.

In this section, a number of systems that perform dynamic updating to various degrees were described and their shortcomings noted. Our primary collective criticism of the systems described is that they are not transparent to the programmer who must use them. Some of the updating systems require the programmer to use a specific language or system to obtain the benefits of dynamic updating while others lack support for distributed computation. We believe such limitations preclude these systems from being used on a wide variety of problems in different application domains.

3. THE BASIC APPROACH TO OUR DYNAMIC UPDATING SYSTEM

In the approach presented here, the procedure-oriented model to dynamic program updating was adopted since many of the programs that could benefit from such a capability tend to be written in procedural languages. For example, portions of the code in a telephone switch [19] and most of the Unix™ operating system (kernel and utilities) are written in C [20]. This research adheres to four main goals.

- to provide a system where a new version of a program can be loaded without stopping and restarting, or significantly degrading the performance of the currently running version,
- to provide a system that is usable with existing languages and scalable to a large-scale (on the order of several hundred computers) distributed environment,

™Unix is a trademark of AT&T Bell Laboratories.

- to minimize the amount of user intervention needed to perform an update in order to allow novice computer users to operate the updating system, and
- to support multiple simultaneous updates, i.e., starting one update before the previous updates complete.

The last goal, multiple simultaneous updates, is essential in a distributed system since an update may take time to propagate through a network. Also, by having this capability, computers in the system can be offline (or can defer an update) when the update is initiated.

Since two of the above goals are to operate on a wide variety of programming languages and be easy to use by a novice, there are cases where the proposed updating system does not work well. As it is assumed that programs are written in a top-down manner, programs that do not follow this requirement cannot be updated by our system. For example, if a program is written as, say, one large procedure, the approach will not work. We believe the increased simplicity of our system gained by not attempting to update such programs is a worthwhile tradeoff.

3.1 Updating Nomenclature

Before discussing some of the updating system details, a precise definition of an update is required. A program Π_{old} is updated to a new version Π_{new} when all procedures P in Π_{old} have been replaced by their corresponding new versions in Π_{new}. A procedure P_{old} is updated when it is replaced by its new version P_{new}. Alternatively, one can view the updating of a procedure P as changing the binding of the name "P" from its "old" implementation to its "new" implementation.

3.2 Synopsis of an Update

Prior to the initiation of an update, the new version of the program is compiled, linked, and loaded into the computer. Thus, the time required to replace an old version of a procedure with its corresponding new version is significantly reduced.

When an update is initiated, the updating system examines the state of the running program and determines which procedures may be updated immediately and which procedures must be updated at a later time. Throughout the update process, the state of the modification is checked periodically and old procedures are updated when all the updating conditions are satisfied (see Section 3.3). In a "well-structured" program, when all procedures have been updated, the program has been updated since it is identical to the new version of the program.

3.3 Criteria for Updating a Procedure

The updating system updates a procedure based on system-computed syntactic criteria and user-provided procedural semantic dependencies. An automatic detection of all semantic dependencies is not possible, if semantic dependencies exist between procedures, the user must provide a semantic dependency list. All dependencies are characterized in the definition of active procedures. Only procedures that are both syntactically and semantically inactive can be updated.

3.3.1 Syntactic Dependencies. Syntactic dependencies are the relationships between procedures in the program that can be ascertained from the program's syntax. In this implementation, syntactic dependencies are detected by the system. Formally, syntactically active is defined as follows.

Let a program Π consist of a set of procedures $P_1, \cdots, P_n$.

Let P be a procedure in Π. Let $\delta^*(P)$ be the set of procedures which are reachable from P in the procedure call graph corresponding to Π. Thus, an invocation of P may result in the direct or indirect invocation of procedures in $\delta^*(P)$. We refer to $\delta^*(P)$ as the syntactic dependency function since it may be calculated from Π's call graph, i.e., Π's syntax.

At any time t, all procedures on the runtime stack are active. In addition, a procedure P is active if its new version P_{new} can call a procedure Q already on the runtime stack, i.e., $Q \in \delta^*(P_{new})$.

A procedure P is inactive when the above criteria for being active are not met.

The active definition and the updating algorithm allow new (updated) procedures to call other new procedures. Old (not updated) procedures may call other old procedures or new procedures via appropriate mechanisms. The primary motivation for this structure is to force programs to be updated from old to new versions. Not only does this correspond to the way the program was developed, i.e., an implicit evolution from old to new, but it also reduces the amount of additional work the programmer must perform to maintain consistency during the update (see Section 3.5.).

3.3.2 Semantic Dependencies. A semantic dependency is a relationship between procedures that is not detectable from the program's syntax, e.g., two procedures work together to perform some task but do not directly reference any of the same entities. The updating system deals with semantic dependencies by using information supplied by the programmer. Semantic dependencies are formally stated as follows.

Let $\delta^*_{up}(P)$ denote the semantic dependency function of procedure P: the set of all procedures Q that must be concurrently updated with P. If Q did not change

between versions, it must still be inactive when P is updated.

If $\exists Q \in \delta^*_{up}(P)$ and Q is active, then P is active.

Based on these definitions, a procedure P can only be updated when it is inactive and all procedures $Q \in \delta^*_{up}(P)$ are also inactive. This allows the programmer to specify procedures that must be updated concurrently, thus allowing semantic dependencies to be accommodated. The updating system updates all procedures $Q \in \delta^*_{up}(P)$ atomically. The detection of semantic dependencies adds an additional check before a procedure can be updated.

Using the syntactic and semantic dependency definitions, a procedure may be updated as follows: A procedure P that has not changed between versions and has no associated semantic dependency should be updated[2] when the update is first initiated. Alternatively, a procedure Q that has changed between versions or is semantically dependent on another procedure Q′ may be updated only when it and Q′ are not active. For the remainder of this paper, an active procedure satisfies both the syntactic or semantic dependency active definitions.

Periodically the updating system rechecks for active procedures becoming inactive. The rechecking occurs whenever the runtime stack contains less elements than it did when the last procedural update occurred. Since an inactive old procedure cannot become active (it will have already been converted to a new procedure), by checking the size of runtime stack after the return of each procedure call, all procedures are updated at the earliest possible update time.

3.4 Justification for the Updating Criteria

Based on the definitions given in the previous section, it is possible to contrive examples in which the proposed updating system would not work. Fortunately, if a top-down programming method [7, 10, 24] is followed, such programs should not occur in practice. In programs developed using a top-down approach, the higher-level procedures specify the algorithms of the program and hence are not likely to change between versions. On the other hand, the lower-level procedures, which describe many of the details used to implement the algorithms, are more likely to change between versions. Consequently, in most cases, an update to a program will complete in a short period of time since:

- much of the program will not change between versions,
- higher-level procedures will tend to be the same between versions,
- lower-level procedures will tend to be less active than the higher-level procedures, and
- much of the code is not frequently executed, e.g., code for exception handling.

3.5 Maintaining Consistency During an Update

During an update, the program may contain a combination of both old and new procedures. To maintain program consistency, specially-constructed procedures, called interprocedures, that map old procedure specifications, i.e., calling sequences and return codes, into new procedure specifications are required. Similarly, special-purpose procedures called mapper-procedures map old static data to the appropriate new representation. These procedures aid in providing an orderly migration path from the old version of the program to the new version. Mapping static data or specifications from new to old is not required due to the bottom-up procedure replacement scheme employed. Thus, no additional mapping procedures are needed.

3.5.1 Mapper Procedures. When a procedure that contains local static data is updated, the updating system invokes a user-written mapper-procedure or mprocedure that maps the static data into whatever representation is required by the new version of the procedure. The mapping operation need only be done once per update for each procedure that requires it. Extending this to encompass non-local data requires that all non-local data to be accessed via abstract data types (ADTs) [1]. Note that this restriction is consistent with the programming style constraints discussed earlier.

By taking advantage of the large address space (see Section 4), the mprocedure programmer can write an mprocedure for a procedure P that copies pertinent data from the old version of P, P_{old} to the new version of P, P_{new}. This mprocedure is invoked by the updating system when procedure P is updated. When P_{new} is invoked for the first time, the state information accumulated by P_{old} will have already been installed into P_{new}.

3.5.2 Interprocedures. Suppose program Π calls a sorting procedure "sort" and sort was initially implemented as a bubble sort. In the new version of Π, suppose that sort is implemented as a Quicksort. The specification, i.e., the parameters passed to and the values returned from sort, has not changed between versions. Instead, what has changed is the implementation of sort. In this case, the updating system, as described thus far, is able to replace the old sort with the new sort. What happens if the specification of sort as well as its implementation has changed between versions? For example, suppose sort previously sorted

[2] In this case, the procedure is not actually replaced but instead flagged as being "new" since its old and new versions are the same.

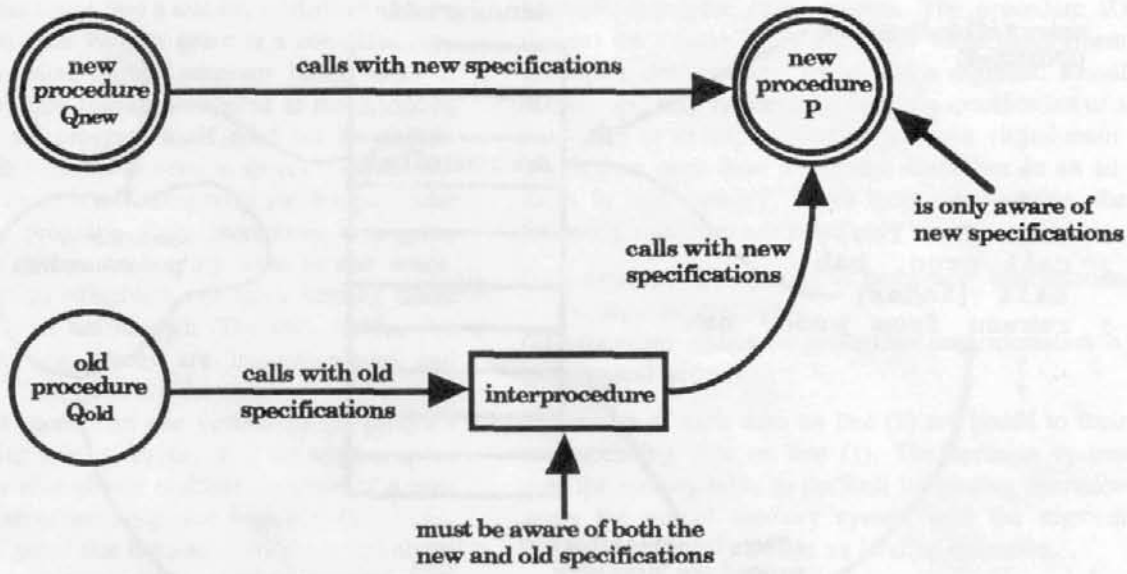


Figure 1. Invocation of an updated procedure P.

arrays of integers but now sorts arrays of real numbers. To properly update sort, a method of converting the specification of sort from one version to another is required.

Converting procedure specifications from old to new versions is similar to converting local static data (Section 3.5.1). Suppose procedure P has been updated as shown in Figure 1. When any procedure Q that has not been updated calls P, it thinks it is calling P_{old} since it does not know about P_{new}. Because P has been updated, P's interprocedure will be invoked instead of P_{old}. The interprocedure will then map the call to P_{old} into an equivalent[3] call to P_{new}. In effect, the interprocedure is creating two illusions. Besides making Q think it is calling P_{old}, the interprocedure is also making P_{new} think that a new version of Q is calling P_{new}. Due to this organization, the interprocedures are aware of the dynamic updating and that multiple versions of a program exist, while the procedures in a particular version of the program are not.

4. THE DYNAMIC PROGRAM UPDATING SYSTEM ARCHITECTURE

In this section, the key architectural design issues of the proposed updating systems are discussed. The section begins with an introduction to the updating system procedure-binding issues and concludes with a discussion of the system memory organization.

[3]In the cases where an exact mapping is not possible, the mapping should be a "reasonable approximation" to the correct call. Determining such a mapping is left to the discretion of the programmer(s) writing both the program and the interprocedures. Since the interprocedures would normally be written at the same time the new version of the program is written, the knowledge required to construct the mapping would be known at this time.

4.1 Binding Tables

In the updating system, the ability to quickly bind and unbind the name of a procedure with its address in a particular version is of paramount importance. In abstract terms, a method of associating a procedure's *name* (the version-independent specification of the task that the procedure performs) with its address (the version-dependent implementation of how the procedure performs its task) is required.

One binding scheme that could be employed in the updating system relies on binding tables and proceeds as follows. A procedure call is performed by indexing the table and branching to the address stored in that table entry. For example, suppose in program Π, procedure foo calls procedure bah. If bah is procedure #2 in Π, 2 is the index into the binding table. The address stored in location 2 is that of bah. This scenario is depicted in Figure 2.

If the target computer's architecture supports indirect addressing, e.g., the Motorola MC68020 [18] or the Intel 80386 [9], the procedure invocation operation can be made quite efficient. In the example given above, a procedure call that uses this type of indirection might look like:

```
CALL [BTBA + 3]
```

where BTBA denotes the binding table base address and the brackets denote the contents of address BTBA + 3. In contrast, an ordinary, i.e., no indirection, procedure call, might look like:

```
CALL bah
```

where bah is the address of procedure bah. The main advantage of using the indirection is that the binding of

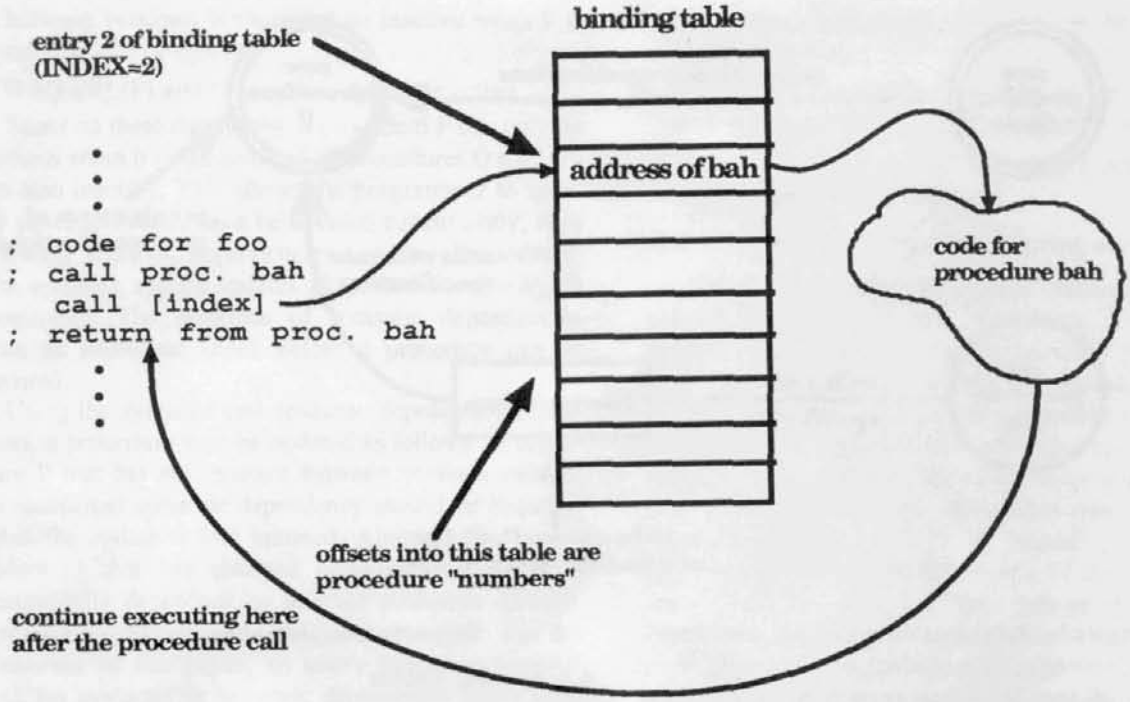


Figure 2. Implementing an updating system with a binding table.

bah can be changed by modifying the value stored at address BTBA + 3. This would be done by the updating system at the appropriate times.

The binding table mechanism described above provides a simple method of procedure binding but has a number of problems. Because the binding table is shared between the operating system and the user program, efficient methods of accessing and protecting the consistency of the binding table must be devised. Another problem with the simple binding table model is that it does not provide a straightforward way of incorporating interprocedures into the updating system.

Consider the following scenario. Suppose procedure P has been updated. Other procedures that have also been updated will attempt to call P with P's new parameters while procedures that have not been updated will attempt to call P with P's old parameters. Because there is only one location in the binding table to store the address the procedure P, either the old procedures or the new procedures will be calling the wrong version of P. To avoid this scenario, the binding table must be multiplexed in some manner.

4.2 Large Address Space Model

To address the above concerns conceptually, the entire updating system rests on top of a large, sparse address space.[4] The addressing mechanisms need not be physically provided by the hardware. The large addresses are partitioned into fields as shown below.

other@version ID@type@proc ID@disp

The "other" field denotes addressing information used by the operating systems such as process IDs, user IDs, or location information. This field can also be used to provide for hardware consistency checking as well as security between processes and machines.

The "proc ID" field denotes the procedure ID of a given procedure. This number corresponds to the procedure number used to index the binding table. The "procedure ID" for procedure P remains consistent throughout all versions of the program. Thus, the procedure ID may be thought of as the internal name of the procedure it represents.

The "disp" field is the displacement within a procedure and is used for accessing code and data within a procedure. By convention, a zero displacement denotes the beginning of a procedure.

The "type" field specifies what kind of procedure proc ID is. A procedure can be a normal procedure, an interprocedure, an mprocedure, or a remote procedure. The use of remote procedures in the updating system is not discussed here. Interested readers are referred to Segal and Frieder [21]. If the notion of "procedure" is extended to data, the type field can also be used to specify if a given proc ID is code or data. This capability can be used to implement a single-level store.

The "version ID" field represents the version number of the specified procedure. The version ID parti-

[4] A similar approach was taken in the Apollo Domain system [16] where a large address space was used to build unique object identifiers, or UIDs, to address every entity in the system.

tions the address space into a number of distinct version spaces. Within each version space is a complete copy of a given version of the program being updated. Because our system manages versions at the operating system level, the program itself need not be directly concerned with versions or version spaces. Within each version space there is a binding table for that particular version of the program. Only procedures in a given version space utilize the binding table in that space. This allows us to effectively multiplex binding tables since each version has its own. The only entities that may cross version spaces are interprocedures and mprocedures. Thus, these are the only entities that need to know about more than one version of the program and the updating system. Partitioning the address space in this manner also allows multiple versions of a program to exist simultaneously (see Section 4.6).

It should be noted that the addresses described above are generated by a combination of addressing information encoded in instructions, information kept in auxiliary registers, and data structures. In particular, if the address space is, say, 96 bits, all 96 bits need not be encoded in the addressing information of every instruction. Similarly, the CPU(s) in the system do not require 96 address lines. The smaller addresses are converted to large addresses by a virtual memory controller (VMC). This type of addressing architecture is described further in Frieder [14].

4.3 Segmented Virtual Memory

The updating system requires that the underlying computer system support a logical, segmented virtual memory [12]. Such a virtual-memory system takes an address of the form

segment descriptor@displacement

and maps it into a physical address (or a virtual address which is passed to a paged virtual-memory system). Information for each process' segments is stored in a segment descriptor table. Each entry in the table contains information such as the segment's base address in real memory, the size of the segment, and the attributes of the segment (is the segment readable, writable, executable, etc.), and any other information that the operating system and architecture requires.

Since this information is different for each process in the system, most architectures have a segment descriptor register that points to a segment table. The register points to different segment descriptor tables depending upon which process is running.

4.4 Mapping the Large Address Space to Segmented Virtual Memory

The procedure ID field of the large address and the segment descriptor described above are quite similar in a number of respects. Both the procedure ID and segment descriptor name objects. The procedure ID denotes the "name" of a procedure while the segment descriptor denotes the "name" of a segment. Recall that the updating system must bind the specification of a procedure to its implementation just as a virtual-memory system must bind a segment descriptor to an address in real memory. From these comparisons, the following equalities are noted.

(1) procedure ID (name) = procedure specification = segment descriptor
(2) procedure address = procedure implementation = physical address

Variants of each item on line (2) are bound to their corresponding item on line (1). The updating system uses the binding table to perform its binding operation while the virtual memory system uses the segment descriptor table to perform its binding operation.

4.5 The Updating System and Segmented Virtual Memory

In the updating system, procedure IDs in programs to be updated are represented by segment descriptors. Since this results in exactly one procedure per segment, the displacements within procedures are identical to displacements within segments. Therefore, segment descriptor tables are binding tables. As the virtual memory architecture supports a segment descriptor table register, by changing this register to point to a different segment descriptor table, we are changing the procedure specification implementation to the bindings of the program. Such an operation is supported by an operating system primitive accessible to the updating system. During program execution, the only time this register changes is when an interprocedure or mprocedure crosses version space boundaries. Because the program being updated does not know about version spaces, no references to the spaces nor the segment descriptor table register need to appear in the program's code. Since ordinary address generation in an executing program does not directly reference the segment descriptor table register, these concepts are compatible. An implementation of the updating system using segmented virtual memory is shown in Figure 3.

By partitioning the space of segment descriptors, the type field of the large address is incorporated into the segment descriptors. Hence, interprocedures and mprocedures are accessed using the same method as normal procedures. Partitioning the segment descriptors this way also enables data to be accessed directly. This allows permanent i.e., file data to be accessed via the single-level store, assuming the operating system provides a method of mapping the data into the address space. Also, mprocedures can use this mechanism to

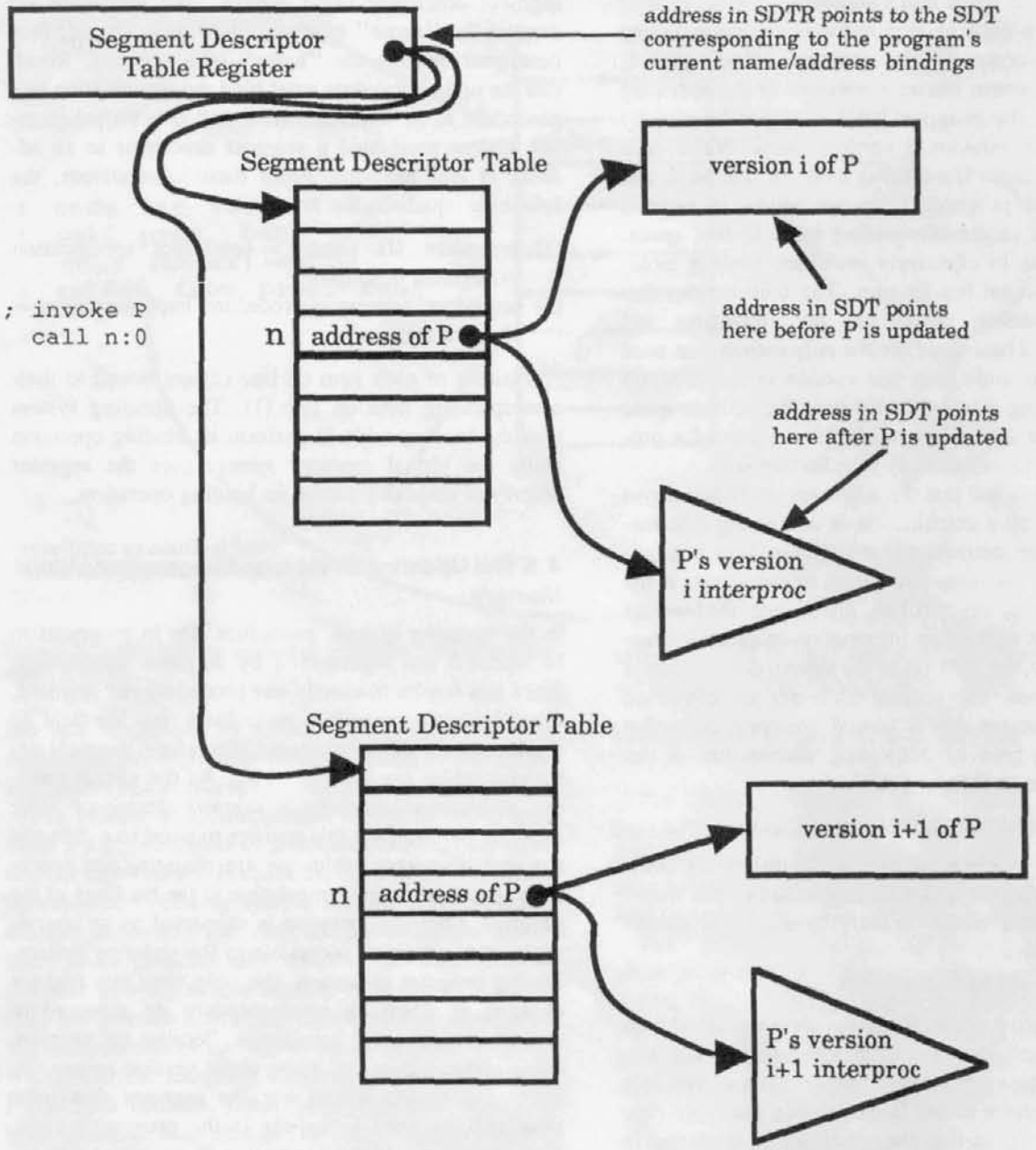


Figure 3. Implementing an updating system with virtual memory.

gain access to data in other procedures for conversion to the new version of the corresponding procedure.

```
i_proc:  ....                          ;manipulate parameters
         SVC INTERCALL,proc,v + 1      ;call new proc
         ....                          ;manipulate return code
         RET                           ;return from interproc
```

This code fragment performs local parameter manipulation, changes bindings to the new version space, invokes the procedure, and returns. Assumed is that the procedure that called the interprocedure is in version v and the new procedure is in version v + 1. The INTERCALL routine denotes an operating system call

When a procedure P is updated, the segment descriptor table entry for P is changed to the address of P's interprocedure since other old procedures need to access P's interprocedure instead of P itself. The machine code for P's interprocedure might look like:

that will cross version spaces and invoke the version v + 1 of procedure proc. INTERCALL must know about both version spaces and adjust the segment descriptor table register accordingly. The INTERCALL routine might be written as follows:

```
INTERCALL:  ....
            PUSH SRD                    ;parameter checking
                                        ;save seg. desc. register
            MOV SDR,TABLE[v + 1]        ;load new seg. table
            CALL_U proc                 ;call user proc in new space
            POP  SDR                    ;reload current version
            RET_U                       ;return to user code
```

This code saves the current segment descriptor table, loads the segment descriptor register with the new version space binding/segment table, and invokes the procedure. When the procedure returns, the segment descriptor register is reset and control is returned to user code. The code to check parameters and manipulate the parameters on a runtime stack is not shown here. Also, CALL_U and RET_U denote routines that perform domain switches to user code before performing the standard CALL or RETURN functions. TABLE denotes a table of segment table addresses. For index x, TABLE[x] contains the address of the version x binding table.

4.6 Multiple Simultaneous Updates

As more than two versions ("old" and "new") can exist simultaneously, multiple simultaneous updates, i.e., starting one update before previous updates have completed, are possible. Having this capability is essential since the time required to update a program is a function of how well it is written. In a distributed system, updating time is also a function of the number of computers in the network. As a result, it may take a long time for an update to propagate to all computers in the network. Similarly, some computers may be offline when the update is initiated. It may also be desirable to defer an update if a computer is temporarily overloaded or if the program running on it cannot tolerate the possible performance degradation that an update might cause.

5. PROTOTYPE IMPLEMENTATION

A prototype of the updating system described in this paper has been constructed. The prototype executes on Sun Microsystems computers running SunOS™ [23] (a BSD 4.3-compatible Unix system) and consists of several major components. The primary user interface to the prototype is called the Updating Shell (ush or "u-shell"). The ush reads commands typed from the user's terminal and, based on the type of command, performs some local action or interacts with other components of the updating system if necessary. The ush is capable of dynamically loading and linking[5] user programs.

User programs are not executed in the same address space as the ush, but rather in the address space of a separate component of the updating system called the Program Update Processor (pup). User programs are loaded by the ush and then downloaded to the pup. The pup and the ush communicate with each other via internet-domain sockets [22], and thus need not reside on the same physical computer system. No direct interaction with the sockets occurs; a communication abstraction reminiscent of remote procedure calls is used instead. This communications subsystem allows messages representing commands to the pup and ush to be interchanged without regard to the idiosyncrasies of sockets. Block data transfers as well as asynchronous message notification are also supported.

The ush can control multiple different programs running on the same host or different hosts. This is accomplished via state-management code in the ush and pup and an additional subsystem (discussed below). Initially, a pup contains no state information. When an ush connects to a pup, state information for the pup's configuration is built and stored in the ush. Upon termination of the ush/pup connection, the state of the pup is transferred from the ush to the pup. As long as the pup (Unix) process is not terminated, the state information is retained. From now on, whenever an ush connects to this pup, the state information is transferred back from the pup to the ush and reconstructed there. This allows the ush user to make and break connections to different pups when desired. The ush may be unconnected from a pup at any time except while an update is in progress.

Each computer in the network may have more than one pup running on it simultaneously. Each pup is referenced by a symbolic program name assigned to it when it is run. The user-specified name symbolizes the

™SunOs is a trademark of Sun Microsystems, Inc.

[5]This feature is not available in standard BSD Unix. It was implemented in the prototype using the standard Unix linker (ld) and a locally-developed loading system.

name of the program that will be loaded into the pup. To arbitrate access to the different pups, a Pup Port Mapper Daemon or pupmapper was constructed. The pupmapper serves a similar function to the Sun Unix RPC portmap daemon. The pupmapper listens on a "well-known," i.e., constant, TCP port for pup mapping commands. When a pup is started, its name is specified as a command-line argument. The pup allocates a TCP port from the available pool on the local machine and passes the tuple [pup name, process ID, TCP port number] to the pupmapper. The pupmapper registers this information for future access requests. When an ush attempts to connect to a pup, it first communicates with the pupmapper. The pupmapper obtains the needed information and returns the tuple to the ush. Additionally, the pupmapper alerts the pup that a user is establishing a connection. By knowing the pup's TCP port number, it can communicate with the pup directly. When a user terminates a pup, the pup removes its information from the pupmapper before exiting. Any future attempts by an ush to connect to this pup will be rejected by the pupmapper.

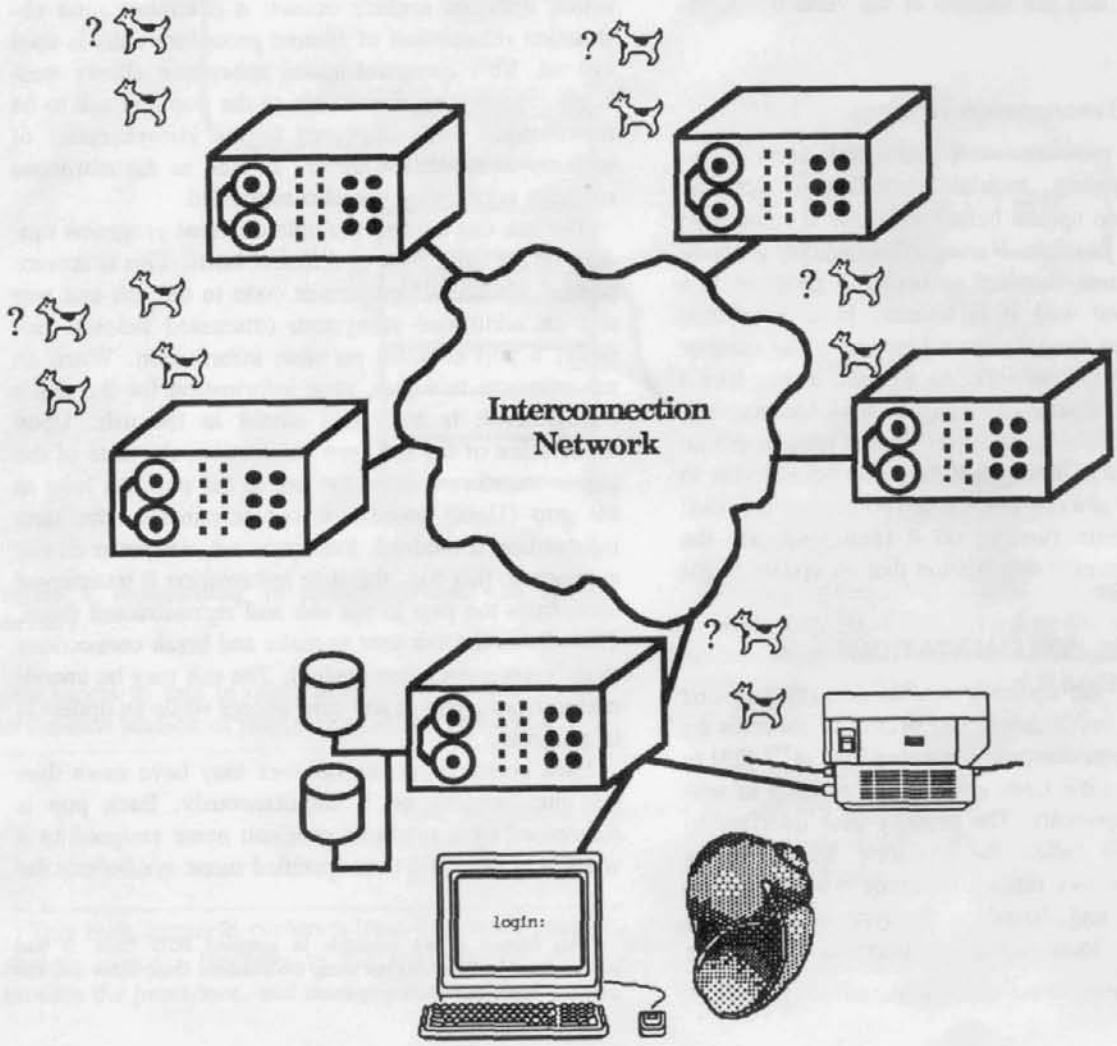
The entire prototype is shown pictorially in Figure 4.

In Figure 4, a number of computers are connected via an interconnection network. Each computer contains peripherals similar to the one shown at the bottom of the figure. At each site there is a single pupmapper (dog with question mark), zero or more pups (dog),and one of the computers has an ush running (shell). The details of the interconnection network are not important here; we merely assume that it is possible (at some level) to get packets from one machine to any other via the network.

Figure 4. A system-level view of the prototype updating system.

6. UPDATING EXAMPLE: NON-STOP PIZZA PREPARATION

The Non-Stop Pizza Preparation Program (pizza) illustrates the events that occur when a program is dynamically updated, as well as some of the capabilities of the



prototype dynamic program updating system. The pizza program controls the hardware of a hypothetical pizza-baking machine. This machine produces fully-baked pizzas by a three-step process. The basic algorithm for pizza may be coded in the C programming language as shown below:

```
main( )
{
  while (1)
    do_pizza( );
}
/* do_pizza drives the procedures that create the pizza.
 * Each pizza is assigned a unique pizza descriptor that
 * is passed to the lower-level procedures. Messages are
 * printed when a pizza is started and when a pizza is
 * finished. The actual parameters used to control how
 * the pizza is constructed are local to the procedures
 * that do the work.
 */
do_pizza( )
{
  static int pd = 1;          /*pizza descriptor*/
  printf("pizza number %d preparation begins . . . \ n",pd);
  set_oven_temp (pd);
  mix_ingredients (pd);
  bake_pizza (pd);
  printf("pizza number %d preparation complete \ n \ n",pd);
  pd + +;
}
```

The main procedure repeatedly calls the do_pizza procedure, in which turn calls set_oven_temp, mix_ingredients, and bake_pizza. One invocation of do_pizza results in the invocation of all procedures necessary to produce one fully-baked pizza. Status messages are also printed as part of do_pizza's pizza-creation process.

The set_oven_temp procedure preheats the oven to a set temperature. This temperature is stored internally to the procedure. When set_oven_temp returns, the oven has been successfully set to the desired temperature and is ready to use. The mix_ingredients procedure measures the appropriate amount of ingredients needed to construct the pizza, mixes the dough, and assembles a pizza. When this procedure returns, the pizza referenced by the pizza descriptor may be baked. The pizza is baked by the bake_pizza procedure. When bake_pizza returns, the pizza is piping hot and ready-to-eat. As with set_oven_temp, both mix_ingredients and bake_pizza use parameters that are stored as local data within the procedures. The set_oven_temp, mix_ingredients, and bake_pizza procedures are shown below.

```
/* set_oven_temp pre-heats the pizza oven to the
 * internally-specified temperature in Fahrenheit
 * degrees. When this routine returns, the oven is
 * ready to cook at that temperature.
 */
set_oven_temp(pd)
int pd;
{
  int temp = 400;
  /* (device-specific pre-heating code using temp) */
  printf("   oven pre-heated to %d degrees",temp);
    printf("Fahrenheit for pizza %d \ n",pd);
}


/* mix_ingredients mixes the internally-specified mass
 * (in pounds) of the ingredients to form a pizza. When
 * this routine returns, the pizza has been assembled
 * and may either be baked or frozen.
 */
mix_ingredients (pd)
int pd;
{
  double dough = 0.75;          /* dough mass (lb) */
  double meat = 0.25;           /* meat mass (lb) */
  double veggie = 0.25;         /* veggie mass (lb) */
  double cheese = 0.50;         /* cheese mass (lb) */
  /* (device-specific mixing and kneading code) */
  printf("pizza %d contains %.21f lb dough,",pd,dough);
    printf("%.21f lb meat,",meat);
    printf("%.21f lb veggies,",veggies);
    printf("%.21f lb cheese \ n",cheese);
}


/* bake_pizza will bake the pizza for the internally-
 * specified amount of time in minutes. When this routine
```

```
 * returns, the pizza has been successfully baked.
 */
bake_pizza(pd)
int pd;
{
  int time = 10;
  /* (device-specific baking code) */
  printf("pizza %d baked for %d minutes \ n",pd,time);
}
```

An experienced C programmer might write pizza differently than shown above; this programming style was chosen for the purposes of top-down design and clarity.

Suppose that pizza has now been put into production use and is baking large numbers of pizza for an ever-increasing customer base. Suppose further that the parameters and algorithms coded in set_oven_temp, mix_ingredients, and bake_pizza produce the thin-crust style of pizza generally eaten in New York City. Unsatisfied with their revenues, the owners of the pizza machine decided to change the program that runs their pizza machine to produce the deep-dish (thick-crust or "stuffed") pizza generally eaten in Chicago. To accomplish this, the set_oven_temp, mix_ingredients, and bake_pizza procedures were altered to produce the new recipe. The do_pizza procedure was also changed to print slightly different status messages. After these changes have been made, the new version of pizza must be installed. Rather than shut down the pizza machine to install the new software (and lose additional revenue), the pizza machine owners decide to dynamically update the pizza program with the new thick-crust version.

6.1 Updating the Pizza Program

Having explained what pizza does and given an overview of the prototype, we now demonstrate how pizza is loaded, executed, and updated using the prototype dynamic program updating system. Selected sample dialogs with the ush and pup are shown below. User input is shown in boldface type.

When the ush is started, the following is displayed:

```
citi% ush
Updating System Shell version 0.0 (compiled on May 15/89 at 02:24)
    running under SunOS version 4.0 on host citi
>
```

When the pup is started on another computer, the following is displayed:

```
gto% pup x
program update processor version 0.0 (compiled on May 14/89 at 03:09)
  running under SunOS version 4.0 for program x
```

To connect the shell started above to the pup, the ush must be given a symbolic name of a program and told which host on the network is running a pup for the desired program. The program name should correspond to the name of the program supplied on the pup (x in the example above). If no host is specified in the connect command, ush assumes that the pup resides on the same host as the ush. Since the ush and the pup need not reside on the same physical computer, it is possible to remotely load, execute, and update programs. In this example, the ush is running on a computer named citi and the pup is running on a different computer named gto.

```
> program x
The current program is now x
> connect gto
Connected to pup for program x on host gto
Pup for program x on host gto is new
>
```

Once the connection has been established, programs may be loaded and run. To load version 1 of pizza, the following is done:

```
> load 1 pizza1.delta pizza1.o
Load sequence begins . . .
  extracted updatable procedures from delta file pizza1.delta
  program x has 5 updatable procedures
  program x's procedures registered with pup name service on host gto
  delta* computed for 5 procedures in x:
      0  1  1  1  1
      0  0  1  1  1
      0  0  0  0  0 · · ·
```

```
        0  0  0  0  0
        0  0  0  0  0
        :
    program object code image requires 1192 bytes
    program loading at address 0x26738 in pup x's address space on host gto
    program linking complete
    loaded VMC segment descriptor table into gto's VMC
    copying code to pup . . . complete
    no interprocedures loaded for version 1
    no mprocedures loaded for version 1
Version 1 of program x loaded successfully
>
```

This display is printed with the full debugging option compiled into the ush. The δ^* table shown above has been shrunk for the purpose of this example. Because there are only five procedures in this program, the rest of the table is zeros. Once the program has been loaded, it may be run immediately or additional versions of pizza may be loaded. Loading version 2 results in:

```
> load 2 pizza2.delta pizza2.o
Loaded sequence begins . . .
    extracted updatable procedures from delta file pizza2.delta
    program x has 5 updatable procedures
    program x's procedures registered with pup name service on host gto
    delta* computed for 5 procedures in x:
        0  1  1  1  1
        0  0  1  1  1
        0  0  0  0  0 . . .
        0  0  0  0  0
        0  0  0  0  0
        :

    program object code image requires 1272 bytes
    program loading at address 0x27548 in pup x's address space on host gto
    program linking complete
    loaded VMC segment descriptor table into gto's VMC
    copying code to pup .. complete

    iproc object code image requires 608 bytes
    iproc loading at address 0x283a8 in pup x's address space on host gto
    iproc linking complete
    copying code to pup . complete

    mproc object code image requires 264 bytes
    mproc loading at address 0x2860c in pup x's address space on host gto
    mproc linking complete
    copying code to pup . complete

Version 2 of program x loaded successfully
Version 1 interprocedures loaded successfully
Version 1 mprocedures loaded successfully
>
```

Although not shown here, it is also possible to load a new version of the program while the current version is executing. Notice that the output indicates interprocedures have been loaded for the pizza program even though none are required. In the prototype, every procedure has an interprocedure and mprocedure associated with it. If the interprocedure or mprocedure is not required, it merely passes parameters without alter-

ing them (interprocedure) or returns without doing anything (mprocedure).

Running version 1 of pizza is accomplished by:

```
> run
Program x is now running
>
```

When the ush is given the run command, version 1 of the current program is executed. On the pup, the following is printed when debugging is enabled (debugging messages are preceded by pup:):

```
pup: INTERRUPT! (pup state: 0) received command RUN
Execution begins at address 0x26738 . . .
pizza number 1 preparation begins . . .
   oven pre-heated to 400 degrees Fahrenheit for pizza 1
   pizza 1 contains 0.75 lg dough, 0.25 lb meat, 0.25 lb
     veggies,
   0.50 lb cheese
   pizza 1 baked for 10 minutes
pizza number 1 preparation complete
pizza number 2 preparation begins . . .
   oven pre-heated to 400 degrees Fahrenheit for pizza 2
   pizza 2 contains 0.75 lb dough, 0.25 lb meat, 0.25 lb
     veggies,
   0.50 lb cheese
   pizza 2 baked for 10 minutes
pizza number 2 preparation complete
        :
```

The INTERRUPT message signifies that the pup has received a RUN command from the ush. This causes version 1 of the currently loaded program to begin execution. As shown above, pizza is creating pizzas and printing out status information for each step.

Suppose we wish to update pizza to version 2 to begin preparing thick-crust pizzas. On the ush side, we type:

```
> update pizza2.changes
Update of program x from version 1 to version 2 begins . . .
    4 procedures have changed between versions:
          do_pizza
          set_oven_temp
          mix_ingredients
          bake_pizza
    program x runtime stack backtrace:
          do_pizza
          main2
    warped 1 unchanged procedures into version space 2
    computed inactive procedures:
          set_oven_temp
          mix_ingredients
          bake_pizza
    warped 3 changed procedures into version space 2
    enabled procedure return interrupts
Program x update initiated successfully
>
```

```
        some time passes . . .
    *** Program x updated to version 2 successfully ***
>
```

and on the pup side, we see (with debugging enabled):

```
        :
pizza number 6 preparation begins . . .
   oven pre-heated to 400 degrees Fahrenheit for pizza 6
   pizza 6 contains 0.75 lb dough, 0.25 lb meat, 0.25 lb
     veggies,
   0.50 lb cheese
pup: INTERRUPT! (pup state: 2) received command
     UPDATE
   pup: UPDATE dispatcher received command GET_RTS
   pup: UPDATE dispatcher received command SET_SDTE
   pup: UPDATE dispatcher received command SET_SDTE
   pup: UPDATE dispatcher received command SET_SDTE
   pup: UPDATE dispatcher received command SET_SDTE
   pup: UPDATE dispatcher received command SET_HRVN
   pup: UPDATE dispatcher received command EI_POP
   pup: UPDATE dispatcher received command EOF
   TC pizza 6 baked for 15 minutes
pup: PROCEDURE INVOCATION TERMINATED!
   pup: UPDATE dispatcher received command GET_RTS
   pup: UPDATE dispatcher received command EOF
pizza number 6 preparation complete
pup: PROCEDURE INVOCATION TERMINATED!
   pup: UPDATE dispatcher received command GET_RTS
   pup: UPDATE dispatcher received command SET_SDTE
   pup: UPDATE dispatcher received command DI_POP
   pup: UPDATE dispatcher received command EOF
TC pizza number 7 preparation begins . . .
   oven pre-heated to 475 degrees Fahrenheit for TC pizza 4
   TC pizza 74 contains 1.25 lb dough, 0.50 lb meat, 0.50 lb
   veggies, 1.00 lb cheese
   TC pizza 7 baked for 15 minutes
TC pizza number 7 preparation complete
        :
```

As before, the lines preceded with pup: are the debugging output of the pup interacting with the ush to perform the update. These lines correspond to the ush's view of the update operation. Looking at the ush output, when pizza was interrupted, it was inside the do_pizza procedure. The ingredients have been mixed but the pizza has not yet been baked. The main and do_pizza procedures are currently on the runtime stack as shown in the previous ush screen display. At this point, set_oven_temp, mix_ingredients, and bake_pizza may be updated since they are not active (Section 3.3). The do_pizza procedure may not be updated since it is active (because it is on the runtime stack).

The actual update is accomplished by the ush sending a number of commands to the pup. Before each command is processed, a pup debugging message is printed. The GET_RTS command causes a runtime stack snapshot to be sent from the pup to the ush. This is used by

the ush to determine what procedures may be updated. This list is printed by the ush. The SET_SDTE command causes the pup to manipulate the virtual memory controller's segment descriptor tables which cause procedure bindings to be changed. This is done at least once per procedure update. The SET_HRVN command causes the pup to set its internal notion of highest running version number. In this example, the update could not be completed when it was initiated. When such a situation occurs, the ush instructs the pup to notify it whenever procedure invocations terminate. After each procedure invocation, the pup asynchronously notifies the ush of this event. This is normally undetectable by the user running the updating system. After the ush has been notified of the procedure termination, it rechecks the program's state to see if the update may be continued. This mode is set by the EI_POP command, which signifies enabling procedure-invocation return interrupts.

After this mode has been set, pizza continues execution. The next step of the pizza process is the baking of the pizza. As can be seen from the pup screen display, this step takes place.[6] When the bake_pizza procedure returns, the pup interrupts the ush. A quick check of pizza's runtime stack reveals that do_pizza is still active and execution continues. When control returns to do_pizza, it prints a message stating that pizza number 6 has been prepared. After this occurs, do_pizza returns control to main. The do_pizza procedure is now inactive and is updated. Upon completion, the ush notifies the pup to disable the procedure termination interrupts by sending the pup a DI_POP command. The interrupts are no longer needed since all changed procedures have now been updated. The ush then prints out a message informing the user that the update is complete. The pizza program now continues to execute its second version. Note that in the second version, the amount of ingredients, temperatures, and baking times are different. All messages are also preceded with the letters TC, signifying "thick-crust" pizza is being made. The pizza descriptor numbers are kept sequential by an mprocedure associated with procedure do_pizza. When do_pizza was updated, this mprocedure copied the value of the pizza descriptor from the old do_pizza into the new do_pizza. If this were not done, the first time the version 2 do_pizza procedure ran, it would start with pizza #1, violating the uniqueness rule of the pizza descriptors.

6.2 The Semantic Dependencies of Baking a Pizza

Although the pizza program appears to have been updated successfully, this is not entirely correct. When pizza number 6 was baked, it was baked for 15 minutes instead of the usual 10 minutes of bake time required by a thin-crust pizza. Notice also that the baking message is preceded by TC. This update is wrong since the pizza was baked too long, thus resulting in a burnt pizza. Even though all changed procedures were updated when they were inactive, something else has gone wrong.

This update did not proceed correctly because a semantic dependency (Section 3.3.2) between the three procedures that control the pizza machine (set_oven_temp, mix_ingredients, bake_pizza), and the procedure that controls when these procedures (do_pizza) are called, was not stated. Although there is nothing in the program's syntax that says a thin-crust pizza cannot be baked for 15 minutes, this information must be given to the updating system. In the prototype this is accomplished by specifying which procedures must be updated concurrently. This allows semantically dependent procedures to be updated as a group, thus maintaining program consistency. If this semantic dependency were correctly specified, the update would have appeared as follows:

```
> update pizza2.changes.sd
Update of program x from version 1 to version 2 begins . . .
    4 procedures have changed between versions:
        do_pizza
        set_oven_temp
        mix_ingredients
        bake_pizza
    program x runtime stack backtrace:
        do_pizza
        main2
    warped 1 unchanged procedures into version space 2
    computed inactive procedures:
    no changed procedures could be warped to version space 2 at this time
```

[6] Note that TC precedes the messages. This change will be explained shortly.

```
        enabled procedure return interrupts
Program x update initiated successfully

 >
        some time passes . . .
        *** Program x updated to version 2 successfully ***
>
.
.
pizza number 10 preparation begins . . .
        oven pre-heated to 400 degrees Fahrenheit for pizza 10
        pizza 10 contains 0.75 lb dough, 0.25 lb meat, 0.25 lb veggies,
            0.50 lb cheese
pup: INTERRUPT! (pup state: 2) received command UPDATE
        pup: UPDATE dispatcher received command GET_RTS
        pup: UPDATE dispatcher received command SET_SDTE
        pup: UPDATE dispatcher received command SET_HRVN
        pup: UPDATE dispatcher received command EI_POP
        pup: UPDATE dispatcher received command EOF
        pizza 10 baked for 10 minutes
pup: PROCEDURE INVOCATION TERMINATED!
        pup: UPDATE dispatcher received command GET_RTS
        pup: UPDATE dispatcher received command EOF
pizza number 10 preparation complete

pup: PROCEDURE INVOCATION TERMINATED!
        pup: UPDATE dispatcher received command GET_RTS
        pup: UPDATE dispatcher received command SET_SDTE
        pup: UPDATE dispatcher received command SET_SDTE
        pup: UPDATE dispatcher received command SET_SDTE
        pup: UPDATE dispatcher received command SET_SDTE
        pup: UPDATE dispatcher received command DI_POP
        pup: UPDATE dispatcher received command EOF
TC pizza number 11 preparation begins . . .
        oven pre-heated to 475 degrees Fahrenheit for TC pizza 11
        TC pizza 11 contains 1.25 lb dough, 0.50 lb meat, 0.50 lb
            veggies, 1.00 lb cheese
        TC pizza 11 baked for 15 minutes
TC pizza number 11 preparation complete
.
.
```

In this case, the pizza that was being prepared when the update was initiated (pizza 10) was correctly baked. Pizza 11 was correctly prepared as a thick-crust pizza. The semantic dependency forced set_oven_temp, mix_ingredients, bake_pizza, and do_pizza to be updated concurrently, thus resulting in correctly baked pizza and happy customers.

7. CONCLUSIONS AND FUTURE WORK

In numerous applications, e.g., air-traffic control, telecommunications, life support systems, etc., disabling the application to update its software is very costly, at times even unacceptable, and such downtime should be minimized. As large-scale software systems with stringent downtime requirements become more prevalent, the scope and importance of this problem will increase.

This paper described the algorithms, constraints, and architecture of a dynamic program updating system. The updating system replaces a running program with a new version without stopping the program. While our system does not require special-purpose hardware or programming-language extensions to work properly, some programming system requirements were imposed. A hardware and software architecture for the updating system was presented. We showed how this architecture can be constructed on top of a conventional segmented virtual-memory system. To verify the updating system concepts, a prototype updating system was constructed. Sample sessions using the prototype to update an example program were given. The prototype was

able to correctly update the pizza program, thus showing that dynamically updating programs written in a conventional programming language (C) by procedure is feasible.

Among some of the topics left for future work are the design and implementation of tools to aid programmers writing interprocedures and mprocedures. Our experiments with the prototype system have shown that constructing these procedures is both tedious and error-prone, and could benefit from appropriate tools. To further examine the viability of our approach, performance data for updates of larger programs must be obtained. We are presently undertaking this work using a packet router as the sample program. Performance and scalability issues in a distributed environment must also be examined.

Because some languages are better-suited to some tasks than others, an updating system should allow programs written in multiple languages, non-procedural languages, such as LISP and Prolog, or concurrent or parallel languages to be updated. Future updating systems should have such capabilities.

Hard real-time programs (programs that must compute results at a given time), by definition, should not be interrupted to install new versions. Unfortunately, no existing updating system (including ours) supports the updating of hard real-time programs. Currently, updating systems do not have a method of processing rules of the form, "Procedure P must be updated in the next 18 milliseconds because it will be needed to meet the next deadline in 34 milliseconds." Although our updating system architecture strives for efficiency, it by no means guarantees hard real-time deadlines. A real-time dynamic program updating system should be investigated, as it would potentially allow more types of programs to be updated.

If an updating system is to play a role in the development and maintenance of future software systems, the ramifications of dynamic updating in software development and maintenance environments must be examined. If dynamic updating can be done inexpensively and quickly, software developers and maintainers might install bug fixes and enhancements sooner than is currently done. This could lead to better software, happier customers, and higher profits. The appropriateness of dynamic updating systems in real environments will be determined by how well they work for a given problem, their ease of use, and the willingness of programmers and managers to incorporate them into their coding and business practices.

REFERENCES

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.
2. S. R. Ali, Analysis of total outage data for stored program control switching systems, *IEEE Journal on Selected Areas in Communications*, SAC-4(7) (1986).
3. A. Birrell and B. Nelson, Implementing remote procedure calls, *ACM Transactions on Computer Systems*, 2(1) (1984).
4. T. Bloom, *Dynamic Module Replacement in a Distributed Programming System*, Ph.D. Dissertation, MIT, June 1983.
5. J. O. Boese and A. A. Hood, Service control point—database for 800 service, IEEE Global Telecommunications Conference, 1986.
6. R. T. Boute, J. DeMan and H. Peeters, Secure on-the-fly software modification, *Proceedings of the IEE Fourth Int'l Conference on Software Engineering for Telecommunications Switching Software*, 1981.
7. F. P. Brooks, Jr., *The Mythical Man-Month*, Addison-Wesley, Reading, MA, 1975.
8. R. P. Cook, StarMod—A language for distributed programming, *IEEE Trans. Software Engineering*, SE-6(6), (1980).
9. J. H. Crawford and P. P. Gelsinger, *Programming the 80386*, SYBEX, Inc., San Francisco, 1987.
10. O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, 1972.
11. C. L. Davis and E. A. Irland, Software reliability and quality, an assessment of the state of the art, IEEE Global Telecommunications Conference, December 1985.
12. H. M. Deitel, *An Introduction to Operating Systems*, Addison-Wesley, Reading, MA, 1984.
13. R. Fabry, How to design a system in which modules can be changed on the fly, *Proc. Second International Conference on Software Engineering, IEEE*, October 1976.
14. G. Frieder, Cooperative module architectures and their underlying operating system, *State of the Art Report on Distributed Systems*, INFOTECH Publications, Berkshire, 1976.
15. H. Goullon, R. Isle, and K. Löhr, Dynamic restructuring in an experimental operating system, *IEEE Trans. Software Engineering*, SE-4(4) (1978).
16. P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson, and B. L. Stumpf, The architecture of an integrated local network, *IEEE Journal on Selected Areas in Communications*, SAC-1(5), (1983).
17. I. Lee, *DYMOS: A Dynamic Modification System*, Ph.D. Dissertation, University of Wisconsin, 1983.
18. Motorola, Inc., MC68020 32-Bit Microprocessor User's Manual, Prentice-Hall, Englewood Cliffs, N.J., 1985.
19. R. F. Rey, (ed.), *Engineering and Operations in the Bell System* (second edition), AT&T Bell Laboratories, Murray Hill, NJ, 1986.
20. D. M. Ritchie and K., Thompson, The Unix timesharing system, *CACM* 17(7), (1974).
21. M. Segal and O. Frieder, Dynamically updating dis-

tributed software: Supporting change in uncertain and mistrustful environments, in *Proc. IEEE Conference on Software Maintenance*, October 1989.

22. Sun Microsystems, Inc., Interprocess communication primer, *Networking on the Sun Workstation*, Mountain View, CA, 1985.
23. Sun Microsystems, Inc., *Unix Interface Reference Manual*, Mountain View, CA, 1986.
24. N. Wirth, Program Development by Stepwise Refinement, *CACM* 14(4), (1971).
25. R. H. Yacobellis, J. H. Miller, B. G. Niedfeldt, and S. S. Weber, The 3B20D processor & DMERT operating system: Field administration subsystem, *The Bell System Technical Journal*, 62(1), (1983).
26. J. Magee, J. Kramer. and M. Sloman, Constructing Distributed Systems in Conic, *IEEE Trans. Software Engineering*, *SE*-15(6), (1989).