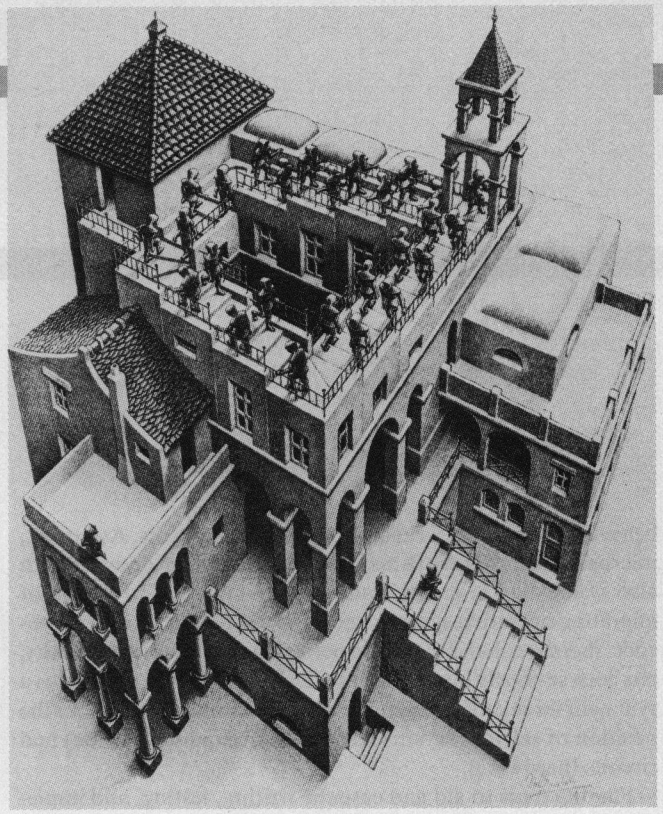

M.C. Escher, *Ascending and Descending*, 1960 © M.C. Escher Heirs c/o Cordon Art, Baarn, Holland

We can achieve more optimal function assignments between microcode and software levels by applying techniques described herein to statically or dynamically microprogrammed processor design.

Automated Vertical Migration to Dynamic Microcode: An Overview and Example

Robert I. Winner, Institute for Defense Analyses
Edward M. Carter, United States Air Force Academy

Firmware engineering provides a method for system tuning—one of its most promising uses. A technique called *vertical migration* moves functions from high-level applications into control store, improving performance appreciably. The term *dynamic microprogramming* describes the loading of a control store with different functions throughout the lifetime of a process (including the operating system). We can have a useful system-tuning environment if we select vertical-migration candidates in an automated manner, and if we migrate them to control store in a transparent manner.

An overview

This article overviews vertical migration and dynamic microprogramming, highlights their benefits, and (concentrating on interfaces between the various involved software elements) describes a general vertical-migration system. We will describe an exemplary automated vertical-migration system called ATOM (abstract type-oriented migration) and will discuss and relate a general dynamic microprogramming management scheme to the example implementation.

Modern computer systems can be viewed as hierarchically arranged interpreters, each level interpreting the next level up and providing it with primitive operations. Using primitive operations found in the macroarchitecture, for example, microprograms interpret machine language instructions.

We shall refer to one skill involved in designing a computer system (that is, proper function placement in specific hierarchical levels) as architecture binding. Such placement requires that system architects (1) make subjective judgments regarding applications to be run on machines and (2) optimize performance for the most likely applications—resulting in architectures tuned for a specific problem class. Unless architects have perfect foresight, the resulting architecture may be unsuited for other problems.

Myers[1] describes this architectural tendency to improperly support many problems as the "semantic gap." One solution would delay binding until the problems to be run are described. Vertical migration, allowing the late binding of an architecture, moves functions from one computer system level to another. Primitives supporting an abstract data type (a queue data structure, for example) might be moved from an application program into a programming language—as in the Ada rendezvous. Primitives might also

0740-7459/86/0700/0006$01.00 © 1986 IEEE

be moved into system microcode, providing new machine level instructions and effectively redefining machine architecture. Colwell[2] discusses some benefits of this migration type.

Reduced-instruction-set computers (RISCs), another contemporary approach to architecture design, provide low-level, language-directed primitives and require compilers to map high-level language constructs onto these primitives.[3] The approach reported in this article has characteristics of both the reduced and the complex instruction set approaches. Vertical migration compiles directly to RISC-like microcode where appropriate, thereby creating a complex "instruction" exactly matching software needs.

Dynamic microprogramming—the alteration of a computer's control store contents while the system is running—requires writable control store (WCS) rather than read-only (ROM). Processors often have conventional machine language emulators in ROM with optional WCS for user microprogramming. However, we don't know of any current commercial system providing more than the most rudimentary WCS support. A few vendors provide so-called Fortran accelerators, microprograms implementing some intrinsic Fortran mathematical functions. While some operating systems may support loading WCS once during cold starts, no commercial product provides serious support for user-implemented dynamic microprogramming.

Vertical-migration performance benefits depend upon whether or not microcode written to solve a specific problem will perform better than conventional generic code aggregated to solve the same problem. In many contemporary machine architectures, instruction sets are designed to solve broad problem classes—a reality influenced as much by marketing as by technical decisions. If the instruction set is too specific, the market is limited and the machine's commercial success is similarly constrained. Microprogramming enables us to place new instructions in the machine instruction set, thereby tailoring the architecture for our specific problem solutions. Tomlinson Rauscher pioneered this approach.[4]

Vertical migration improves performance by reducing both fetch/decode overhead processing for machine level instructions and the number of executed microinstructions. Without vertical migration, we must implement functions by a machine level instruction sequence with each instruction fetched from memory, decoded by underlying hardware or firmware, and interpreted by a series of microinstructions. Fetch/decode performance improvement from vertical migration results from performing the normal fetch/decode/execute cycle only once for the entire function. This cycle is required only for the machine level instruction that causes an entry into WCS. Therefore, instead of a memory fetch series, we have only one; and instead of decoding many machine level instructions, we decode only one.

We can further improve performance with a tailored instruction set. Holtkamp[5] describes the overhead of interpreting a general-purpose machine instruction set. This can involve additional microinstructions allowing many different addressing modes, condition code settings, and operands. A tailored instruction set avoids code added for generality because tailored code provides specific solutions—we neither need nor desire generality. Holtkamp also observes that control store entry/exit overhead, expensive in a general-purpose machine language instruction set, can be easily overcome in a tailored vertical-migration environment.

If the microarchitecture provides for parallelism in microinstruction execution, as in architectures known as horizontal machines, we improve performance even more. In these machines, microinstructions are simply collections of micro-orders with each micro-order controlling separate, concurrent functional processor units. For example, we can perform fixed and floating-point arithmetic operations, address generation, and memory access concurrently. Packing micro-orders as densely as possible, with no contention for devices or operands, is the key to performance improvement here. Such microinstruction packing is called compaction. When we migrate functions into microcode, the microinstruction sequences we would have to execute to interpret conventional machine language can almost always be compacted or otherwise optimized.

All vertical-migration, dynamic microprogramming systems seek to increase processor-bound workload performance. To accomplish this, we require the following steps:

- Application programs must be analyzed to determine highly profitable migration candidates;
- The best set from among those candidates must be selected;
- The selected set of candidates must be translated into microcode;
- Macro- and micro-objects must be linked into memory images;
- Micro-objects must be archived;
- Microimages must be associated with running processes;
- WCS must be managed as a virtual store; and
- All of this must be accomplished with minimum overhead and user effort.

A model vertical-migration system

Vertical-migration systems consist of processors and data sets. Processors implement various parts of a vertical-migration policy. A general model description follows, and the next section illustrates our ATOM implementation of this model.

The model. Given a debugged software system implementing some functions, vertical-migration systems must do two things: (1) decide what software parts would migrate most profitably and (2) migrate and translate those parts into micro-objects. The interface between vertical migration and dynamic microprogramming amalgamates micro-objects into usable structures. Dynamic microprogramming causes those structures to be used at the right moments.

These usable structures are called microimages (see Figure 1). Microimage creation is sensitive to both vertical-migration and dynamic microprogramming policies; consequently, creating microimages from micro-objects can be viewed as an interfacing between the two systems.

Figure 1 illustrates, in coarse steps, software progress from ver-

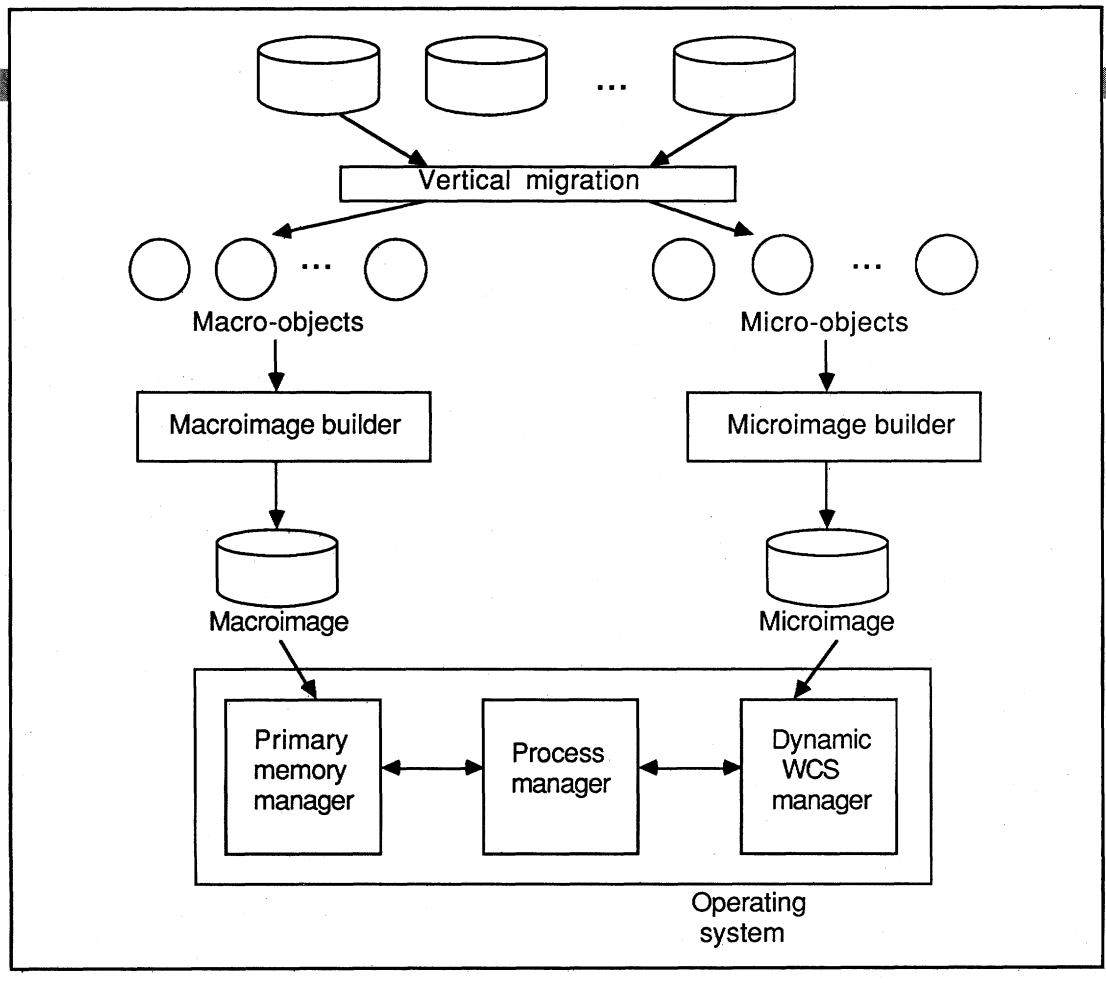

Figure 1. A vertical migration to dynamic microprogramming system.

tical migration to the execution point. Vertical migration splits software objects between macro- and microlevels. Image builders correspond roughly to linkage editors—they are sensitive to vertical-migration policy in the way they choose objects to link, and are sensitive to the operating system's dynamic microprogramming in the nature of produced images.

Profitability. Profitability decisions that vertical-migration systems take must be based on several factors. A set of candidates exists; the system scrutinizes this set and assigns a profitability indication. The resulting set may be larger than the original. This first profit is an object's "local" profit. It describes profit the system predicts will be realized in one execution of that object compared to macrolevel execution. It ignores overhead in getting the object into WCS and interaction with other macro- or microobjects.

New candidate sets may be larger than originally due to granularity of analysis. One basic part of vertical-migration policy is the conceptual object size that can be migrated. Example grains are arbitrary machine language sequences, intermediate code statements, intermediate code sequences, high-level statements, high-level sequences, high-level subprograms, and abstract data types. If grain size for analysis is smaller than macro-object size, then new and smaller objects may be extracted for migration. Thus, assigned candidate sets may be larger than original sets.

While the choice of granularity as related to workload remains to be investigated, all listed approaches have shown significant performance improvements when used in manual and semiautomatic migration experiments. In the granularity trade-off (it seems) the finer the resolution, the smaller the probability of wasting WCS space—but the more complex the profitability analysis. In addition, larger grains such as subprograms and abstract data types could yield information about object relationships that could, in turn, help predict profits.

For example, we can predict local profit by comparing expected individual instruction execution times in a given object's macro- and microforms and taking into account loops and branches. Global profit prediction that must later consider object relationships, however, might benefit from the fact that object A is an abstract data type derived from object B.

Scope of analysis. Another policy issue in the profit analysis phase is scope of analysis. We may intend a single microimage to be loaded when the system is bootstrapped, and used until the system is shut down—as with conventional instruction set emulators and Fortran accelerators. In such cases, to determine microimage makeup requires that scope of analysis be over a large collection of programs.

On the other hand, suppose we want language-specific microimages; there might be a Pascal microimage,[6] for example, or one for each language implemented on a machine such as the Burroughs B-1700.[7] Again, the scope of analysis covers a large collection of programs.

Yet another possibility exists, that the scope be over a small class

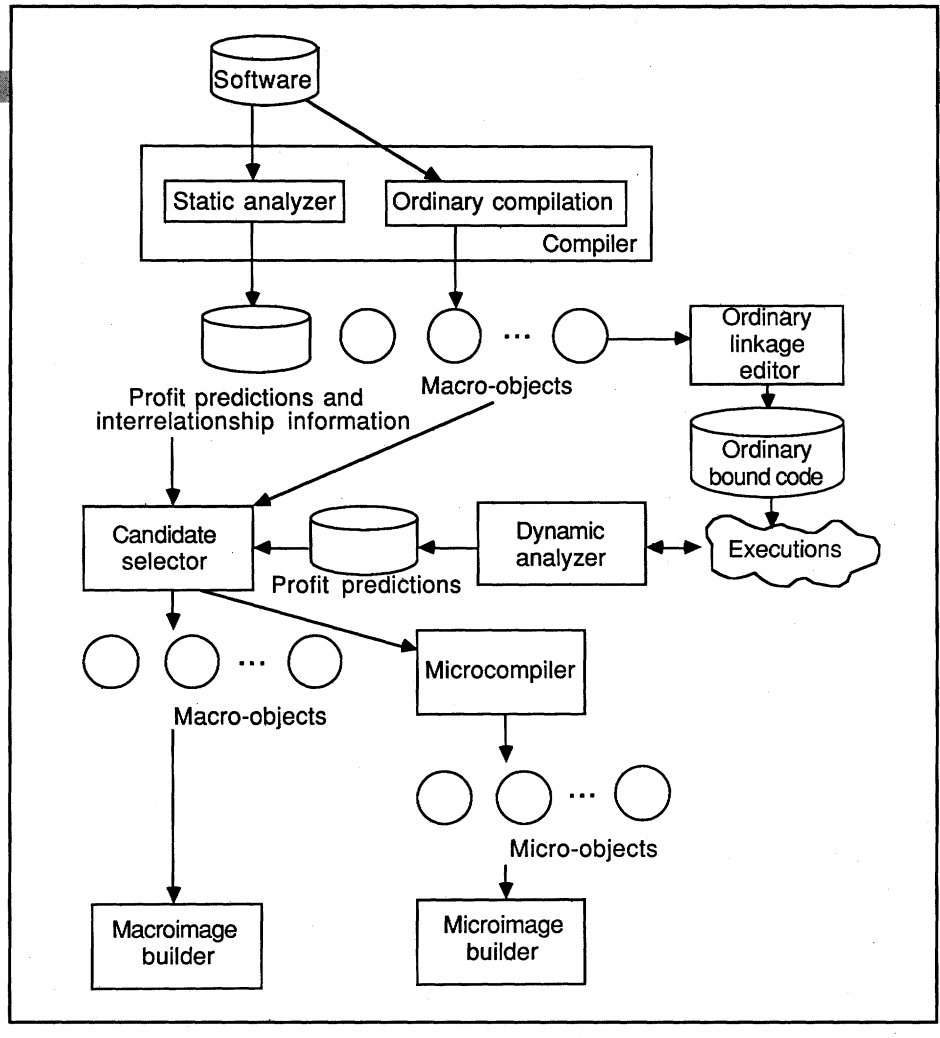

Figure 2. Analysis, selection, translation, and image building.

of programs—such as some processor-bound library package or single application program. The ATOM experiment proposed that scope of analysis should cover the lexical visibility scopes of a block-structured language like Ada. Here, several microimages could exist for a given program. Different processes simultaneously executing this program might each be using different microimages. Analyzing over scopes could capture the fact that optimal microimage selection may depend on input data.

The scope of analysis choice could be critical in total system performance because dynamic microprogramming systems must manage more if scope is smaller. However, we will describe a means of getting around this in the microimage selection strategies section. For the time being, consider analysis to result simply in a set of objects and associated local profits such that any object could in principle be translated either to macromachine or micromachine code.

Analysis. We now turn from profit prediction policies to mechanisms. Two analysis types can be applied to the problem: static and dynamic. In static analysis, we analyze a program's source or intermediate code to find relative processor-use intensities of the various parts. In dynamic analysis, we translate code into conventional machine language and execute with performance monitors in place. Dynamic analysis, consuming more time than static analysis, is more complicated to manage. Static analysis is an internally complex problem. Both approaches can err due to data dependencies. Some evidence shows static analysis quite effective, so its use is worth considering.

Having assigned predicted profits to candidate objects, we need a policy and mechanism for candidate selection and a mechanism for translation to microcode. Figure 2 illustrates relationships between analysis, selection, and translation.

Candidate selection. Candidate selection policy depends on the WCS management scheme in use. We assume that a microimage must fit into physical WCS; that is, WCS is not paged. Otherwise, candidate selection must be driven by a working-set analysis.[8] Assuming we have a fixed, maximum microimage size, we can split candidate selection into two parts: profit revision and image packing. Profit revision inputs objects and their local profits, as well as relationship information that must be produced during the analysis phase. The type derivation relationship mentioned at the end of the profitability section exemplifies this information; its use exemplifies policy applied to revise local profits.

Image packing means fitting as much profitability as possible into an image—the problem is that objects vary in size. Packing images optimally is an NP-complete problem; therefore, we need knapsack heuristics. In our system, we used a greedy algorithm. Holtkamp and Wagner[5] used a more complex heuristic successfully, but in a far more restricted scope of analysis.

Candidate selection can be complicated greatly because profit revision and image packing are not independent. For example, suppose objects A, B, and C have profits that depend upon whether neither, one, or both of the other objects is migrated. Sup-

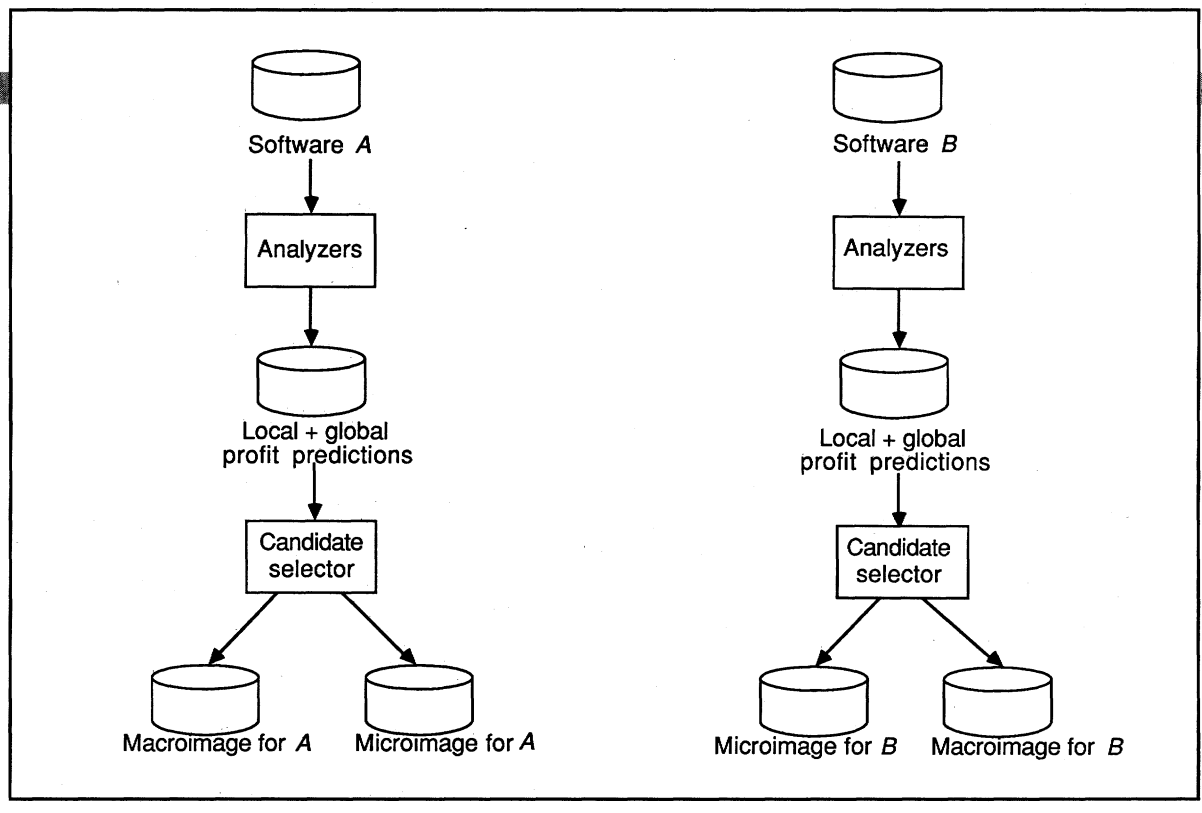

Figure 3. The best-possible-microimage (BPM) strategy.

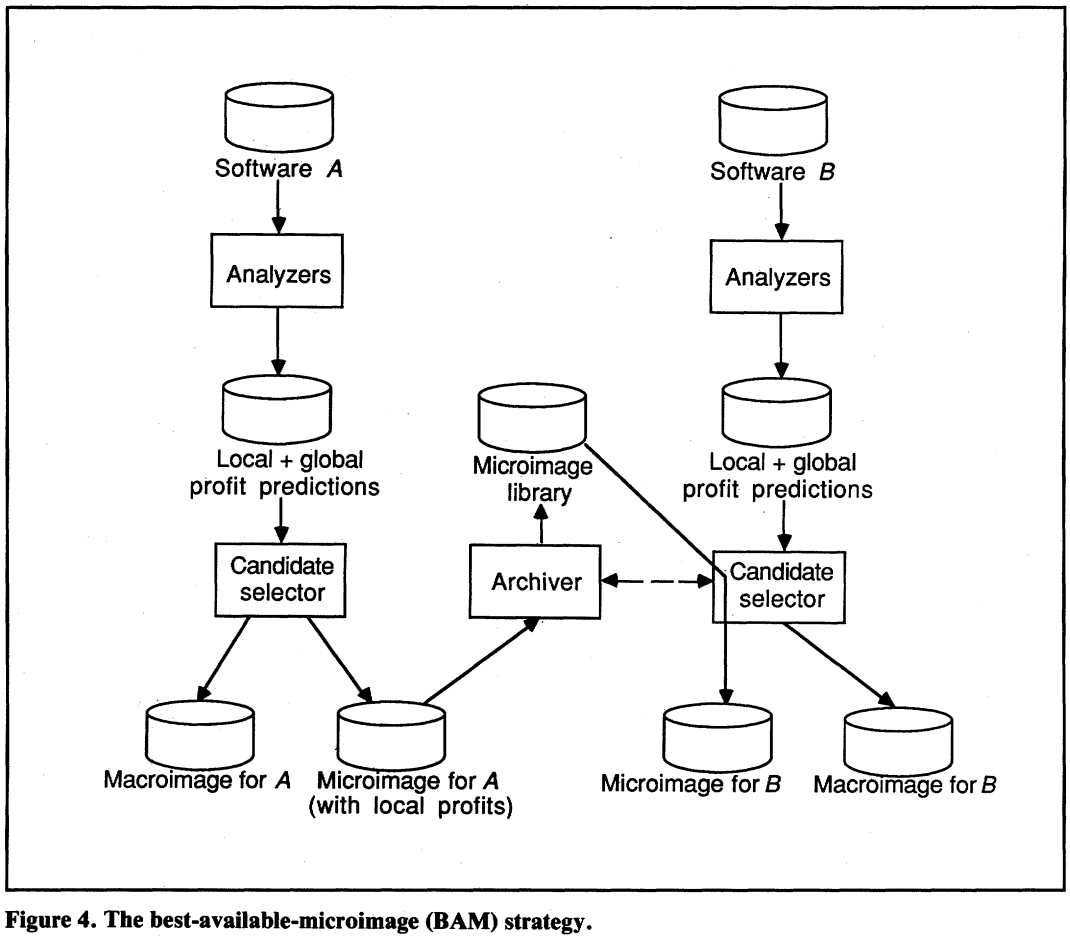

Figure 4. The best-available-microimage (BAM) strategy.

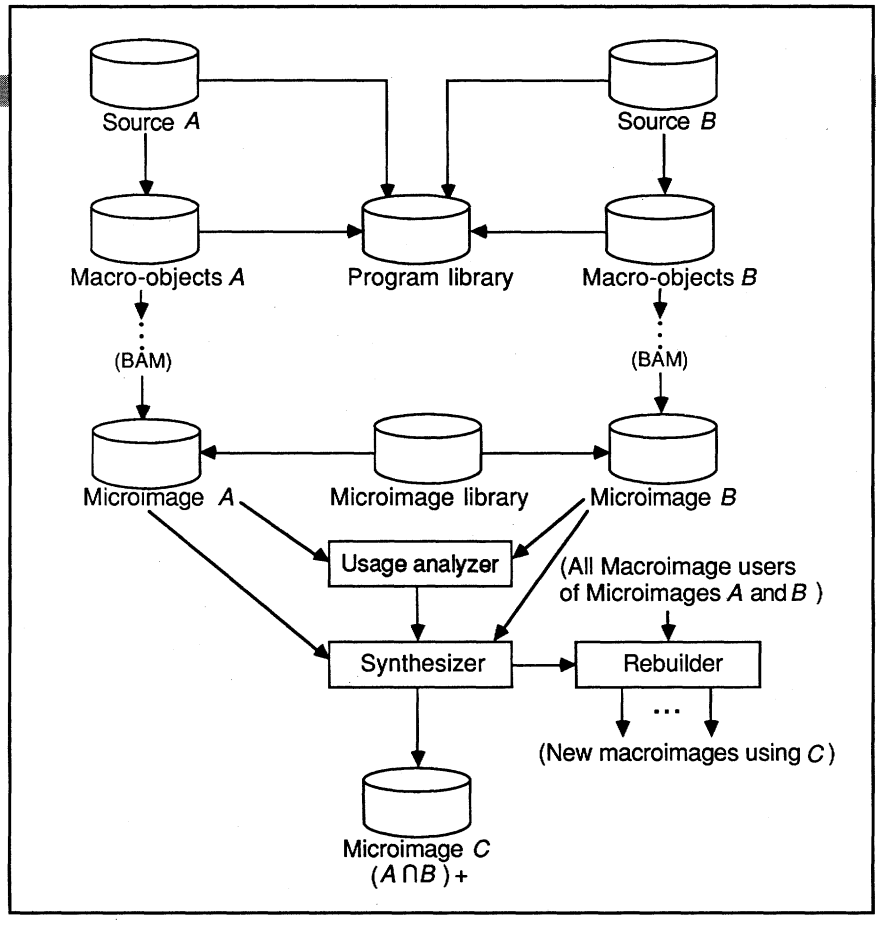

Figure 5. BAM plus microimage synthesis.

pose, however, that only room for two of these exists. The computational effort needed to make this kind of selection mounts quickly as the number of involved candidates increases; furthermore, we don't fully understand how interrelationships affect profits. Nevertheless, fairly simple approaches work reasonably well in the cases we have seen.

Best-possible and best-available selection strategies. We have glossed over a fundamental issue: When should *global profit* come into play? Global profit represents the advantage gained by the complete microimage when used with a particular macroimage. Thus, a micro-object's local profit is weighted by how frequently the macroimage uses it. If we desire the optimum microimage for a particular macroimage, global profit predictions must affect the microimage candidate selection policy—meaning that an optimum microimage be built from available micro-objects when the macroimage is built. We call this the best-possible-microimage (BPM) strategy, the method of choice for engineering special-purpose computer systems (see Figure 3).

However, we are considering general-purpose multiprogrammed systems and are faced with a possible trade-off between individual process performance and overall workload performance. Taking system-wide performance into account, we can build and archive microimages based on local profits or based on global profits of a sample program set. Microimage selection will then proceed based on the best-available-microimage (BAM) strategy, as Figure 4 illustrates.

The BAM strategy can be elaborated by analyzing microimage usage. We can construct a microimage synthesis process to produce new microimages for retrofitting to existing macroimages (see Figure 5). For example, suppose we use two microimages. We might find common micro-objects in these microimages. We might join the two microimages to produce a new microimage containing the intersections of the originals plus some remaining routines. The two user process classes now become one class, decreasing the overhead of managing WCS. Taken to its logical conclusion, one could synthesize an optimum microimage set for a known system workload (perhaps a singleton). Previous vertical-migration experiments including ATOM implement BPM. However, our vertical-migration linkage editor[9] implements BAM with a BPM override. To our knowledge, no one has implemented microimage synthesis.

Abstract type-oriented migration

ATOM, a vertical-migration system, uses the previous section outline to demonstrate automated-vertical-migration feasibility. For candidate selection granularity, ATOM uses abstract data types. It translates all abstract types into microcode and stores them in microimages based on compilation unit rather than dynamic microimage building. C source files, identified as abstract data types, are then passed through programs extracting information from symbol tables and creating gateways and microimages for process-loading at dispatch time.

Steps involved in ATOM are

- Data collection—the collection of data about the program's execution behavior;
- Candidate selection—the determination of execution environment and types to include in each microimage;
- Image loading—the placement of a call to each new environment at entry into the source program's appropriate lexical scope;

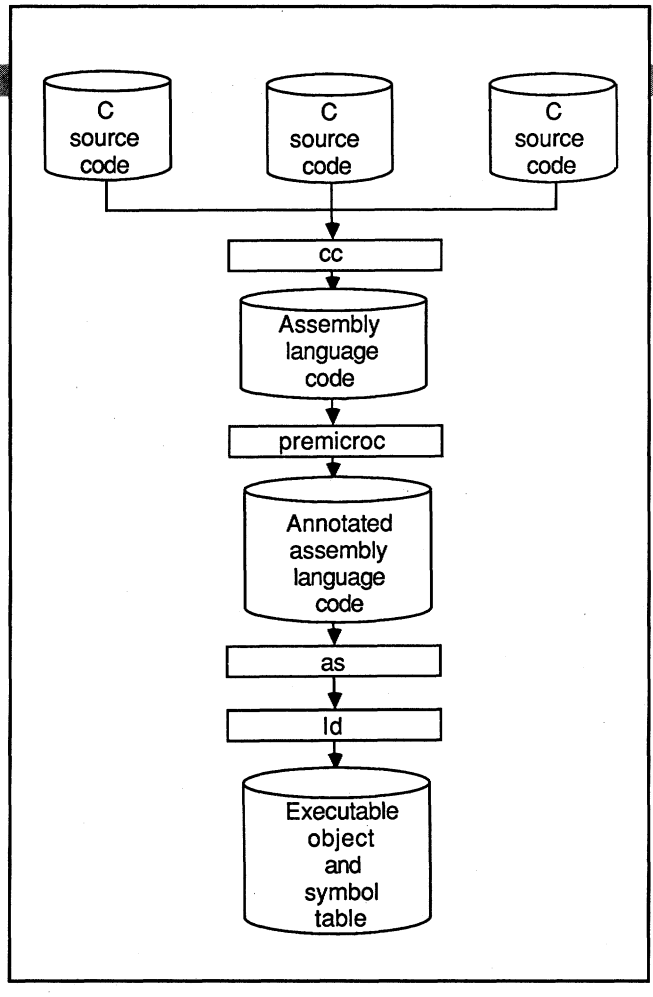

Figure 6. Source program analysis and translation.

- Source program analysis and translation—the program's compilation using an extended C compiler, saving the created microcode source file; and
- Microimage creation—the creation and linkage of a control store image for each execution environment.

We will now detail each of the above steps, showing how they fit into the general vertical-migration scheme. Our references provide a more complete ATOM analysis.[10]

Data collection. Earlier, we described local and global profit determination as very difficult. ATOM simplifies the process by allowing programmers to influence decisions based on performance information collected during previous program executions. A data collection model was developed in ATOM, extracting performance data from instrumented programs and presenting collected data in usable form. Data, oriented towards basic code segments as created by the compiler, was summarized by abstract data-type operations as implemented by C functions. Included were a control-store-size estimate assuming automated migration and comparative execution time estimates resulting from migration to microcode. The data collection program described local profits as determined by a performance estimation model. This analysis, dynamic in nature, is open to data dependency error.

Candidate selection. The candidate selection phase seeks to determine the best set of candidates for migration based on global profit, migrated microcode size, and control store limits. As previously noted, one consideration when performing vertical migration is scope of analysis. In ATOM, we determine this scope statically by the program's lexical structure. ATOM premises that every lexical scope of a block-structured program defines a boundary containing abstract data types that may be defined and used. Each inner scope can have more data types than its enclosing scope and therefore can include more migration candidates. Since some inner scopes may not define new abstract types, we can combine some scopes for simpler analysis. In ATOM, a nested lexical-scope collection sharing the same abstract data types is called an *execution environment.*

Image loading. Another task identified for dynamic microprogramming systems involves loading the microimage once candidates have been selected. The ATOM compiler inserts a call to a routine named "set_mu" at each execution environment entry. This routine's parameters are the name of a microcode image to be loaded and a unique number identifying the entered environment. "Set_mu" calls the operating system to associate the named microimage with the calling process, establishing the passed environment identifier as the current execution environment. We do not load actual images until the macroprogram attempts the first control store entry.

The ATOM programmer associates a microimage with a process, but the compiler could do this with a call to "set_mu" in all but the initial execution environment. An enhanced version of the normal Unix start-up code loads the initial microimage. Microcode translation software (discussed below) supplies the image name. A special device driver, implemented specifically for dynamic microprogramming research,[11,12] manages the control store as a virtual resource.

Source program analysis and translation. We used as much normal Unix software as possible in ATOM, including the C compiler (cc), linkage editor (ld), and assembler (as). Figure 6 depicts first-phase source program analysis and translation. Code analysis identifies abstract operation entry points (C functions with names) replacing normal machine language instructions with a gateway that determines if the operation has been migrated; if so, the gateway performs an ENTER CONTROL STORE (ECS) instruction; otherwise, normal machine language code is executed. Note that the decision to enter control store or the macrolevel implementation is dynamic.

Simple assembly and linkage editing follow code analysis, causing the linkage of micro- and macro-objects into memory images. We need a macrocontext object that maps macronames to primary memory locations and a microcontext object that maps micronames to WCS locations. Code analysis and translation create the macrocontext object for the main program and pass information through the symbol table, creating the microimage and its associated microcontext object.

Microimage creation. ATOM's last phase creates, links, and binds the final microimage to the main program. Figure 7 shows the second stage of language translation and microimage creation. In Figures 6 and 7, premicroc, microc, mas, mulnk, and mumki are software tools built for our research. Also, the Unix linkage editor (ld) was extensively revised for reasons described in the section on dynamic microprogramming. Microc, taking an executable machine language program as input, creates microassembler source code to implement functions identified by premicroc. Iden-

tification information is passed to microc through the executable program's symbol table.

Microc results in a microassembler source file and executable macroprogram that will be associated with the microimage at dispatch time. A series of three programs perform actual microcode generation; the first two (mas and mulnk) are a microassembler and microcode linkage editor, respectively; the third (mumki), a microimage maker, archives the created microimage and stores it for later control store loading.

Dynamic microprogramming

Since the previous sections dealt with vertical migration and candidate selection for migration, we will now assume that objects intended to run in microcode have been chosen, translated, and amalgamated into microimages. Dynamic microprogramming subsystem duties are

- To keep microimages safe from alteration and available for use;
- To cross-link macroimages and microimages;
- To manage active microimages (those associated with live processes); and
- To manage WCS.

For clarity, we must first describe objects of concern.

Objects and programs of particular interest. An unusual aspect of creating runnable images from cooperating microimages and macroimages appears in image cross-linking. To understand this, notions of sharing and context must be detailed. Dynamic microprogramming management performance depends upon how often WCS must be loaded. A required WCS load is called a *WCS fault,* analogous to a virtual-memory fault.

WCS-fault frequency hinges on how much macroimages share their microimages. If all macroimages share one microimage, we find no faults. If each macroimage has a unique microimage, and if microimages are used continuously, a fault will occur every time the scheduler dispatches a process.

If macroimages are to share microimages, we must link each image to its partner at some time; consequently, we must establish a context for the macroimage's view of its microimage (the microcontext) and another context for the microimage's view of the macroimage (the macrocontext). If the macroimage contains a jump to a migrated procedure, for example, that procedure's location is not known at compile time and must be filled in later by a linkage editor. This binding of references to objects establishes the macroimage's microcontext.

The same thing must happen in the other direction. Thus, we can think of contexts as tables mapping names to locations. We refer to tables existing as unified objects (and used in that form at run time) as macro- or microcontext objects. In certain situations, we must manage context objects separately from their associated images.

Another object of interest is WCS itself, which has only a few important characteristics as far as its virtual-device management is concerned. Continue to assume that WCS is swapped and not paged; in addition, assume that there are no instructions for loading WCS from primary memory and for jumping to WCS from primary memory. The former should be a protected instruction and the latter a user-mode instruction.

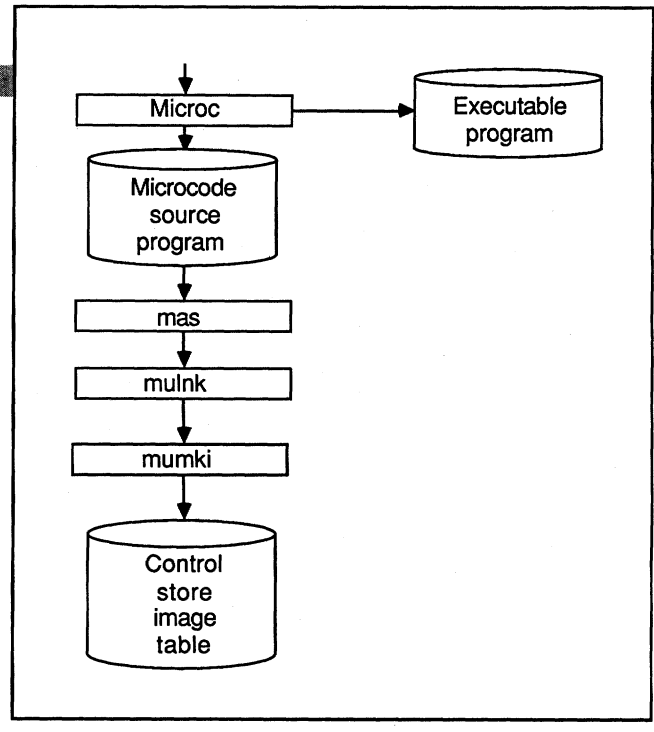

Figure 7. Microcode image creation.

ECS *p,l* means jump to control store physical entry point *p* with parameter *l* in a reserved register. The *physical entry points* are a few reserved WCS locations; four will suffice for normal calls, normal returns, returns from interrupt handling, and a microcode debugging aid. We use parameter *l* to indicate the routine being called; therefore, *l* can be viewed as a *logical entry point.* The physical entry routine can range-check this parameter in microcode. The fewer physical entry points the better, because they must be managed separately from the remaining image and might need frequent reloading.

The last object of interest, the microimage archive, is simply a well-protected micro-object and microimage library containing directories to assist in micro-object selection during microimage building (BPM and BAM strategies) and in microimage selection (BAM strategy). The librarian, as archive manager, might implement extra security measures.

Programs comprising the dynamic microprogramming management system manipulate the above images, contexts, WCS, and archives. These programs—the linkage editor, the microimage activator, the microimage loading and WCS management system, and the microimage librarian—relate as shown in Figure 8.

Cross-image linkage editing. The linkage editor joins objects into an executable program. Of particular interest to us, binding macroimages (shared by several processes) with microimages (shared by several macroimages) becomes a complicated problem whose solution depends on allowed microimage sharing—a problem further complicated by our desire to make microimages invisible to users. Thus, the system must perform all work related to microimage selection and use. In fact, to describe the various designer options and system rationale requires a longer article than this.[9]

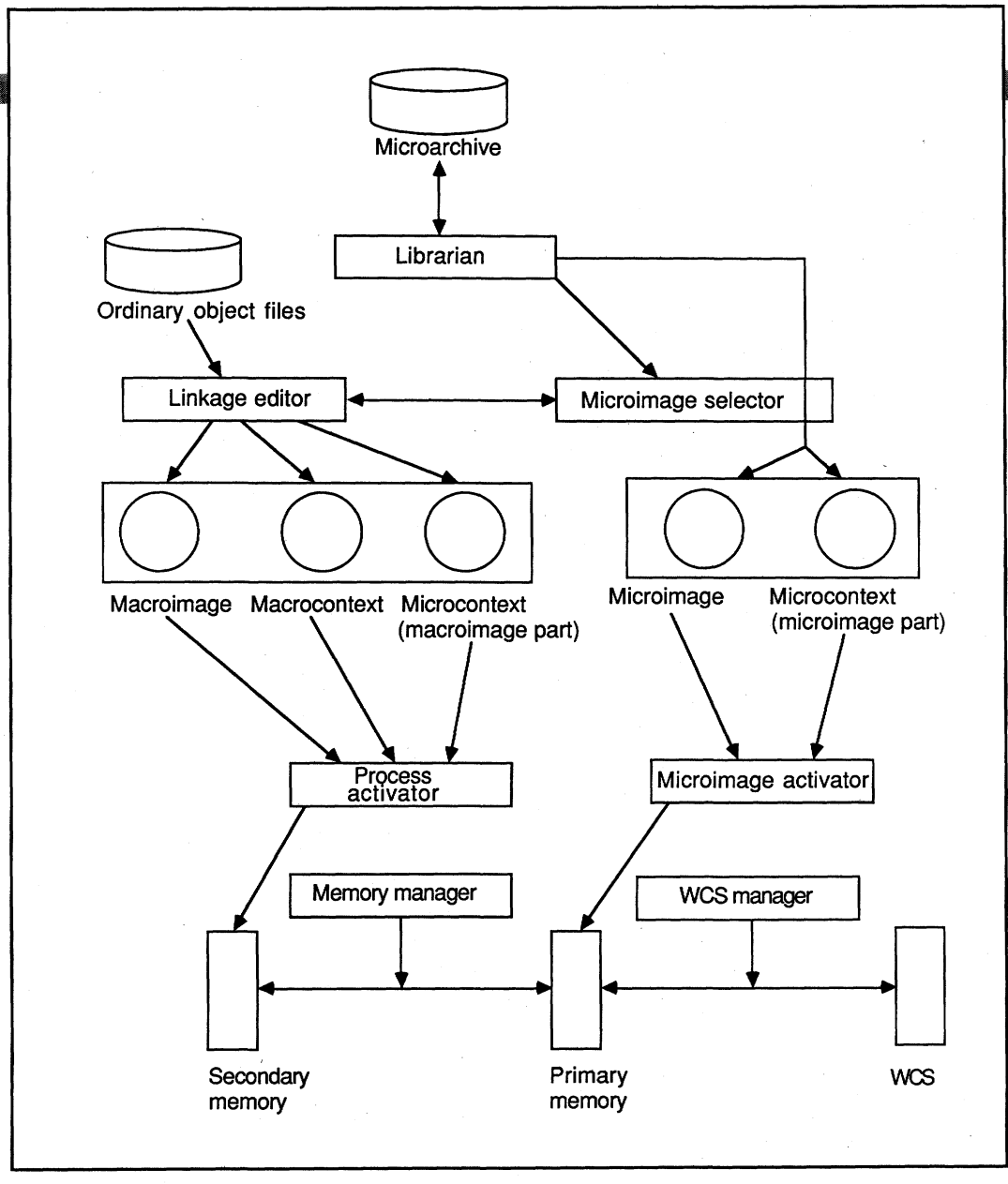

Figure 8. Dataflow of the dynamic microprogramming system.

Among available options, assume that

- Microimages can be shared by different macroimages;
- If a macroimage uses a microimage, then every macro-object referred to by the microimage must be present in the macroimage;
- Every macroimage uses only one microimage at a time and these microimages are determined at linkage edit time;
- A macroimage needs no more than one macrocontext; and
- Migrated objects are subprograms with names.

The first assumption means that each macroimage must have its own macrocontext, perhaps one for every microimage it uses. The fourth assumption implies that one macrocontext will suffice for every macroimage. The second assumption implies that all macroimages using a particular microimage can use all the microimage's routines and, therefore, can have the same microcontext. In particular, a given microimage's table of logical entry points remains constant despite the associated macroimage and, therefore, can be part of the microimage as described in Figure 8.

In addition to its normal tasks, the linkage editor

- Causes microimage selection;
- Inserts the microimage file name into the macroimage start-up routine;
- Includes, in the macroimage, all macro-objects referred to by the microimage;
- Creates the microcontext's macroimage portion; and
- Creates the macrocontext for the microimage.

None of this requires altering the microimage in any way—it remains in the microarchive until activated. The microcontext's macroimage portion usually consists of gateway routines, macrosubprograms with entry points whose names are identical to those of the original migrated objects. These are linked in the usual fashion with macroimage external references.

Creating the macrocontext for use by the microimage(s) is not so obvious. One might mimic ordinary macro-to-macro linking actions, involving the *address-stuffing* technique: The editor literally fills in each reference with the referent's location. Two fac-

tors confound address stuffing, however: First, in many micromachines the microinstruction format does not allow a full primary memory address—addresses must be built in a register by a routine taking several microinstructions. Second, and an important reason for avoiding address stuffing, is that address stuffing embeds the macrocontext in the microimage. Therefore, since all macroimages using a given microimage must have the same context, they must have all static data (including subroutine entry points) in the same locations.

In practice, address stuffing implies that all macroimages using a given microimage must be copies of the same executable file. This is unnecessary. For different macroimages to share a microimage, each macroimage must have its own macrocontext object for use by the microimage. All microimage references to macroobjects are compiled as indirect references via the table in the macrocontext object. On some machines, this executes faster than address-stuffed code.

Microimage activation. The microimage activator locates a named microimage, establishes this microimage in virtual WCS, and connects the client process to the microimage. Providing a user-level system call that includes the desired microimage's file name is an easy way to initiate microimage activation. The activator then employs internal system utilities to resolve user-supplied names to physical file names.

At this point, the activator must detect from the file description that a microimage is valid and whether it is sharable. If the sharable microimage is already in use, the activator merely does housekeeping. Otherwise, the file must be loaded into primary memory (used as a backing store for WCS). Although not proven, it may be important to protect active microimages from secondary memory swapping. We provide such protection by managing active microimages in kernel memory space or in pages locked into primary memory. The operating system must also call the activator when a process creates a new process. The child process inherits the parent's microimage unless a new program file is executed.

Microimage and microcontext loading. Once we have activated the microimage, the remaining issues are how and when to load microimage and microcontext. As to how, we recommend adding an I/O driver to the operating system kernel (Roskos and Winner[11] examine such a driver). A write call on this driver should move a block from primary memory to an indicated location in WCS. The operating system can enforce access rights.

When to load the microcontext and microimage is not so simple: First, the microcontext's microimage portion has two parts, the physical entry points (PEPs) and the logical entry points (LEPs). If these are truly separated, the LEPs are merely contained in a dispatch table and can be permanently attached to the remaining microimage.

PEP management is driven by the need to prevent erroneous WCS entries, to detect WCS faults, and to allow correct WCS entries. To prevent erroneous entries, PEPs join every process context—even those not using WCS. For the abstainers, the PEPs flow to microcode causing an illegal instruction trap. To detect WCS faults (that is, if WCS does not contain the correct microimage when a process is given use of the CPU), the PEPs must also be set to cause a trap handling the fault.

If a process is dispatched, with the correct microimage already present, incorrect PEPs might remain in WCS because the previous process did not use WCS and was from a different class of processes. In such cases (correct image, incorrect PEPs), PEPs might as well be loaded with the correct set during the context switch. If image and PEPs are both correct, we need do nothing.

This results in the microimage of a given process being loaded only at the first execution of an ECS during a process scheduling period. Thus, a process currently in a nonusing phase will not cause microimage traps even though it might be a heavy WCS user during some other phase—a policy resembling demand paging in a virtual-memory system. An option would be to load the microimage at context switch time—resembling virtual-memory prepaging. A third approach would be to load during context switch if the process during its previous scheduling period had executed an ECS—resembling a working-set policy but requiring extra overhead for every ECS.

The worst-case overhead of all these policies depends on the scheduling quantum (the grain of scheduling periods) and the time required to service a WCS fault. On a Perkin-Elmer 3220 running Unix edition 7 with a 200-ms scheduling quantum, worst-case overhead is about 2.25 percent. This requires a very processor-bound set of processes, each using a unique microimage almost constantly; every scheduling period must be one quantum, and every quantum must contain a fault. Therefore, worst-case overhead relates inversely to quantum size, and is also linear in the maximum microimage size. If WCS size increases, worst-case overhead becomes unacceptable.

Clearly, sharing is quite important. If workload includes much sharing, the WCS fault rate will depend on scheduling policy and the number of processors. This interplay—sharing, scheduling, and multiprocessing—is not well understood. Reed and Winner[12] report preliminary results; further analysis and simulation studies are underway. A complete description of dynamic microprogramming management systems, just outlined in this article, can be found in Winner's chapter of *Microprogramming and Firmware Engineering*.[13]

Our intent has been to demonstrate that automated vertical migration through dynamic microprogramming is a useful tool. Practical, automated vertical migration has been proven a feasible instrument in system tuning. This article illustrates how vertical migration can be automated and used transparently in an integrated-system environment. We have not commented on the expected performance improvement level, feeling that final determinations should be reached through experimentation; however, we experienced 70- to 150-percent

improvement in processor-bound program speed (that is, 41- to 60-percent reductions in runtime) using techniques described herein—improvement gained on a vertical microarchitecture with a rudimentary microcode synthesis program.

Perhaps the most relevant consideration still before us is how to implement candidate selection. We can significantly narrow candidate sets in systems like ATOM; however, we implement only a best-possible microimage strategy. We don't attempt workload-optimized microimages.

We can also apply techniques described in this article to statically microprogrammed, special-purpose processor design. In this way, we can achieve more optimal function assignments among hardware, microcode, and software levels. The Institute for Defense Analyses has begun studying automated engineering of special-purpose computer systems. Vertical-migration technology may be fundamental in an environment supporting automatic-system-level computer engineering. □

References

1. G.J. Myers, *Advances in Computer Architecture,* Wiley-Interscience, New York, N.Y., 1982.
2. B. Colwell, "The Performance Effects of Functional Migration and Architectural Complexity in Object-Oriented Systems," PhD dissertation, Carnegie-Mellon Univ., Pittsburgh, Pa., 1985.
3. D.A. Patterson and D.R. Ditzel, "RISC I: A Reduced Instruction Set VLSI Computer," *ACM SIGARCH Computer Architecture News,* Vol. 9, No. 3, May 1981, pp. 443-445.
4. T.G. Rauscher and A.K. Agrawala, "Dynamic Problem-Oriented Redefinition of Computer Architecture via Microprogramming," *IEEE Trans. Computers,* Vol. C-27, No. 11, Nov. 1978, pp. 1006-1014.
5. B. Holtkamp and P. Wagner, "An Algorithm for Selection of Migration Candidates," *ACM SIGMICRO Newsletter,* Vol. 15, No. 4, Oct. 1984, pp. 140-146.
6. M.T. Schaefer and Y. Patt, "Improving the Performance of UCSD Pascal via Microprogramming on the PDP-11/60," *ACM SIGMICRO Newsletter,* Vol. 14, No. 4, Oct. 1983, pp. 140-148.
7. E. Organick and J. Hinds, *Interpreting Machines,* North-Holland, New York, N.Y., 1978.
8. R.I. Winner, "Adaptive Instruction Sets and Instruction Set Locality Phenomena," *ACM SIGARCH Newsletter,* Vol. 11, No. 3, Mar. 1983.
9. R.I. Winner, "Naming and Binding in a Vertical-Migration Environment," IDA Paper P-1938, Institute for Defense Analyses, Alexandria, Va., May 1986.
10. E.M. Carter, "Abstract Type-Oriented Dynamic Vertical Migration," PhD dissertation, Vanderbilt Univ., Nashville, Tenn., Dec. 1983.
11. J.E. Roskos and R.I. Winner, "Toward Sharing the Microprogramming Level on the Perkin-Elmer 3220," *ACM SIGMICRO Newsletter,* Vol. 12, No. 4, Dec. 1981, pp. 67-73.
12. L.B. Reed and R.I. Winner, "Operating System Support for User Microprogramming in UNIX," *Software Practice and Experience,* Vol. 14, No. 12, Dec. 1984, pp. 1183-1196.
13. S. Dasgupta and S. Habib, *Microprogramming and Firmware Engineering,* Van Nostrand-Reinhold, New York, N.Y., scheduled Jan. 1987.

Robert I. Winner is deputy director of the Institute for Defense Analyses' computer and software engineering division in Alexandria, Virginia. His research interests are in the intersection of computer architecture, programming languages, and operating systems. A native of Asheville, North Carolina, Winner earned his BS in mathematics from Union College (Schenectady, New York), his MS in computer science from Purdue, and his PhD in computer science from Georgia Tech. Prior to joining IDA, he was an associate professor of computer science at two universities. He is a member of the ACM, Sigma Xi, and has served on the IEEE-CS TC-Microprogramming board of advisors.

His address is the IDA/CSED, 1801 North Beauregard Street, Alexandria, VA 22311.

Edward M. Carter is a major in the United States Air Force and serves as an associate professor of computer science at the United States Air Force Academy. He received his BS from the USAF Academy in 1974, and his MS in computer science from UCLA in 1975. He attended Vanderbilt University following two Air Force assignments, completing his PhD in computer science in 1983. His primary research interest is the automated adaptation of computer architectures in support of programming languages through dynamic microprogramming.

His address is the Department of Computer Science, USAF Academy, HQ USAFA/DFCS, Colorado Springs, CO 80840-5701.