them as if they are regular instructions and then to perform the analogous optimizations. For example, just as a redundant instruction can be removed or an invariant instruction hoisted out of a loop, so, too, a redundant or invariant check can be removed or hoisted. A simple example is in **Figure 6.21**. Here, the same object reference (held in p and r) is used for a series of getfield and putfield operations. If the first putfield (p.x = ...) passes the null check, then the getfield using the same reference will as well. The null check property can also be propagated to reference r. Thus, only the first null check for p is needed, and the later checks for p and r become redundant. After the redundant null checks are removed (right-hand side of the figure), then the putfield to q. x can be rescheduled after the putfield to r. x.

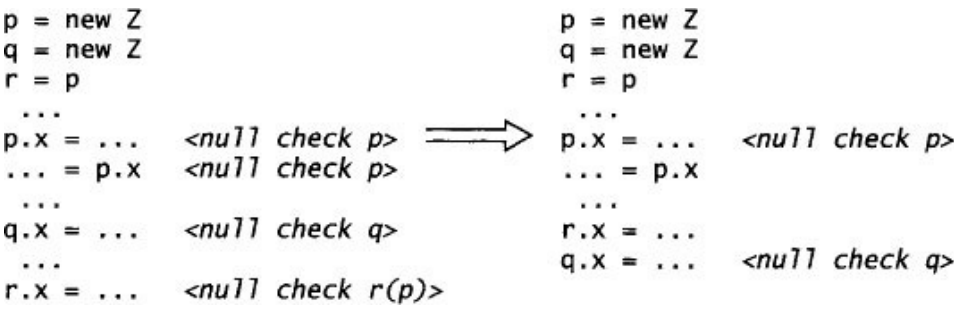


Figure 6.21 Removing Redundant Null Checks. *After redundant null checks are identified and removed, optimizations involving code motion are enabled.*

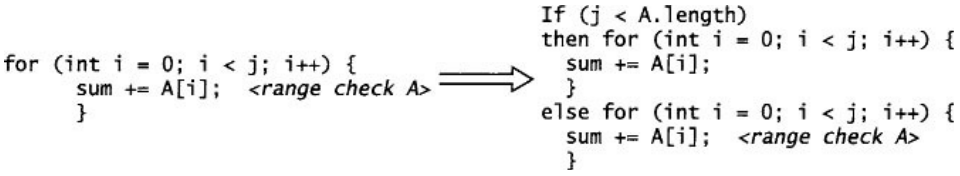


Figure 6.22 Hoisting an Invariant Check. *An array range check is hoisted outside a loop. Then in the common case, no inner loop range check is required.*

An example of hoisting an invariant check is in **Figure 6.22**. In the code on the left, an array range check is performed every loop iteration. However, the check can be hoisted outside the loop (right-hand side of the figure) and only needs to be performed once. After the check has been hoisted outside of the loop, it takes the form of a guard for two versions of the code, one that does not perform the

check and the other that does.

As a final example, a technique known as *loop peeling* can be performed to avoid null pointer problems. In **Figure 6.23** the loop body contains an object reference with a null check. Hence, code motion around the reference is inhibited. However, the first loop iteration can be peeled (right-hand side of the figure); if this first iteration does not cause a null pointer exception, none of the other iterations will. Hence, the null check is not needed for the remaining loop iterations, and other code optimizations can be enabled.

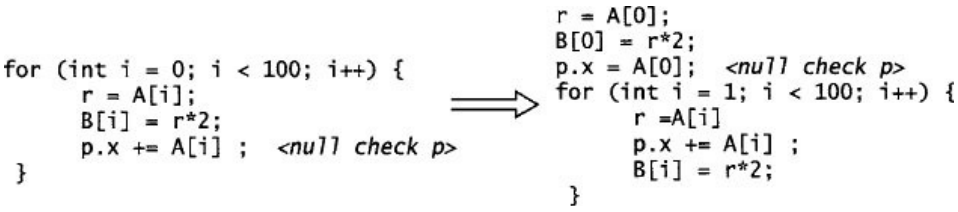


Figure 6.23 Loop Peeling. *By peeling the first loop iteration, the null checks in the remaining loop iterations can be removed.*

Optimizing Garbage Collection

Garbage collection is a key part of a high-performance VM implementation, and there are a number of ways the compiler can provide assistance to the Java runtime to enhance garbage-collection efficiency. First, there may be times when the heap state is temporarily inconsistent, for example, when object references are being modified. At these points, to avoid errors, garbage collection should not be initiated. Consequently, the compiler can provide the garbage collector with “yield point” at regular intervals in the code. At these points a thread can guarantee a consistent heap state so that control can be yielded to the garbage collector. Also, the compiler can help with specific garbage-collection algorithms. For example, if a generational collector is used, then write barriers must be provided by the compiler.

6.7 Case Study: The Jikes Research

Virtual Machine

The Jikes research virtual machine (RVM) was developed at IBM Research and has been made available to the general VM research community (**Arnold et al. 2000**). Jikes is based on an earlier research effort, Jalapeno, and was initially described in research papers using that name. The discussion here is based on descriptions of both the Jalapeno and the Jikes systems.

The overall optimization strategy is to compile only; there is no interpretation step. First, there is a baseline compiler that translates bytecodes directly into native code. This compiler performs no register allocation *per se;* rather, the generated code simply emulates the Java stack. Then for optimization, there is a dynamic compiler that supports three levels of optimization, invoked in stages, depending on an estimate of cost–benefit.

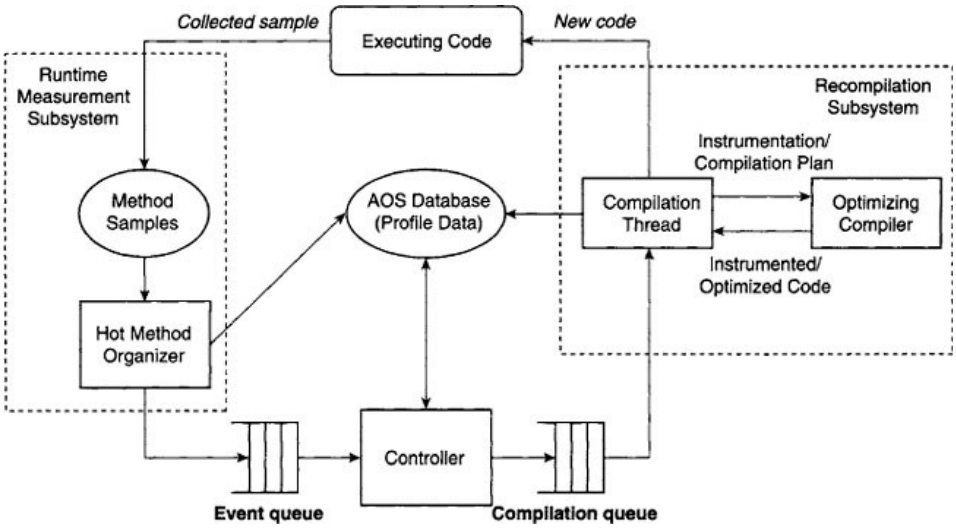


Figure 6.24 The Jikes Adaptive Optimization System (AOS).

Jikes is a multithreaded implementation, with threads responsible for optimization running concurrently with executing program threads. Jikes multiplexes Java application and runtime threads onto threads supported by the underlying host platform, e.g., AIX

pthreads. To allow preemptive thread scheduling, the compiler places yield points, typically at method prologs, epilogs, and loop back edges. At a yield point, the code tests a control bit; if the thread scheduler wants to preempt a thread, it will set the thread's control bit. At the next yield point, the thread will call the thread scheduler.

The overall architecture of the Jikes adaptive optimization system (AOS) is shown in **Figure 6.24**. The major subsystems are the runtime measurement system, the recompilation system, and the controller. These subsystems all interact with the AOS database, which contains profiling data and a history of optimization decisions.

The runtime measurement subsystem gathers raw performance data by sampling at yield points. That is, when a thread switches, a back-edge yield point increments an activity count for the method containing the edge. If it is at a method prolog, the activity count for the calling method is incremented; if it is at a method epilog, the activity count for the called method is incremented.[2] The system can also support other types of compiler-inserted instrumentation to perform edge profiles, basic block profiles, or value profiles, depending on the optimizations that are under consideration. Performance data is initially kept in raw form. Periodically, the raw method samples are analyzed by a separate thread, the hot method organizer, which summarizes and puts the profile data in a form more easily used by the rest of the optimization system.

The controller is responsible for coordinating the activities of the runtime measurement and the recompilation subsystems. It instructs the measurement subsystem to initiate, continue, or change profiling activities. When a method is sampled, that is, found to be active during a thread switch, the hot method organizer places it on the event queue for the controller to consider for recompilation. The controller then makes decisions regarding the recompilation, e.g., whether recompilation of the method should be performed and, if so, at what optimization level.

An important part of the controller's function is to determine whether a method should be recompiled at a higher optimization level. To do this, Jikes uses an analytical cost-benefit model. Say that a given method *m* is currently compiled at optimization level $i < N$, where level *N* is the highest optimization level. Then the controller estimates (1) *Ti,* the expected time the program will spend executing method *m* if it is not recompiled; (2) *Cj,* the cost (time required) of recompiling method *m* at level $j \geq i$; (3) *Tj,* the expected execution time the program will spend in *m* after recompilation. Note that a recompilation at the same level, *i,* is considered because there may be more profiling data available than what was available the previous time *m* was compiled at level *i;* this additional profiling data may change the particular optimizations performed at level *i.* Then the controller determines the optimal level *j,* where $Cj + Tj$ is minimized. Recompilation is performed at level *j,* unless, of course, $Cj + Tj \geq Ti$ and no recompilation is done.

Although this cost–benefit analysis seems very straightforward, the real difficulty comes in estimating the *C* and *T* values. Consequently Jikes uses a combination of heuristics and experimentally derived parameters. To arrive at an estimate of the *T* values, the controller assumes that at its current optimization level, the method will consume as much execution time in the future as it already consumed; i.e., that $Ti = Tcurrent$. The *Tj* values are determined by first estimating the speedups of the different optimization levels using off-line benchmarks. The speedup of optimization level *k* versus level 0 is *Sk*. Then $Tj = Ti * Si/Sj$. Finally, the recompilation cost *Cj* is estimated to be a linear function of the size of the method to be recompiled. The multiplying constant is again determined by off-line benchmarking. If the controller decides that recompilation should be done, it places the pertinent information into the recompilation queue, to be serviced by one of the compilation threads.

The recompilation subsystem consists of a number of threads that

The recompilation subsystem consists of a number of threads that remove compilation plans from the compilation queue. Because they run as separate threads, they can execute concurrently with the Java application threads. A compilation plan consists of the optimizations that should be performed, profile data that will be used by the optimizer, and an instrumentation plan, i.e., the profiling instrumentation the compiler should put in the code it generates (for potential future optimizations).

The AOS database contains a history of profile data and the compilation plans, status, and history of recompiled methods. The controller and recompilation subsystem can query this database to guide further recompilation decisions and optimizations.

The Jikes compiler has three levels of optimization. A brief summary of the optimizations performed at each level is given next.

Level 0: Includes many of the conventional compiler optimizations, such as copy and constant propagation, common subexpression elimination, dead-code elimination, branch optimizations, and several others. So-called *trivial* methods are inlined—these are methods where the code in the method is smaller than the method's calling sequence. In addition, some simple code relayout is performed, and register allocation follows a simple linear scan algorithm.

Level 1: Because many of the classical optimizations are done in level 0, the additional optimizations deal with higher-level code restructuring—there is more aggressive inlining, based on profile information, and there is more aggressive code relayout.

Level 2: Uses a static single assignment (SSA) intermediate form, where each register variable is assigned a value only once (**Cytron et al. 1991**; **Cooper and Torczon 2003**). Using this form allows a number of global optimizations (or more effective versions of conventional optimizations, such as common-subexpression elimination). It also includes some optimizations that can lead to significant

code expansion unless judiciously applied (with the benefit of profile data). For example, loop unrolling replicates the body of a loop, thereby increasing opportunities for other optimizations and eliminating a number of loop-closing branches.

The developers of Jikes have published a number of benchmark results. We give some of the results for the SPECjvm98 benchmark suite. There are seven programs with class files ranging from 10KB to nearly 1.4MB. Performance results are given to measure "start-up" performance and "steady-state" performance. There are two differences between start-up and steady state. First, the steady-state data sets are approximately an order of magnitude larger than the start-up data sets. Second, each steady-state benchmark is run for five iterations by the same JVM instance and the best iteration performance is reported. The start-up behavior will be influenced much more heavily by initialization of data structures, class file loading, and compilation overheads. The steady-state performance is closer to what one might expect in a more realistic production environment, where the start-up overheads have more time to be amortized.

For comparison, a number of compilation scenarios are benchmarked and compared with the simple baseline compilation. In one set of scenarios, the compiler with each of the three optimization levels is run as a JIT compiler. That is, the first time a method is invoked, it is immediately compiled at the given optimization level and remains there throughout the program's execution. The AOS implementation uses all optimization levels in a staged manner, and optimization is performed on a per-method basis. With AOS + FDO (Feedback Directed Optimization), more complete profiling information is maintained, so better inlining decisions can be made and optimizations can be better targeted within methods.

Performance is evaluated as speedup over the simple nonoptimizing

baseline compiler. Figure 6.25 shows start-up results for each of the benchmarks and the harmonic mean speedup for all seven. For the JIT compilers, the lower the optimization level, the better the performance (compress is the only notable counterexample where JIT 1 outperforms JIT 0). For the start-up benchmarks, the extra time required for optimizations cannot be amortized, due to the relatively short run times (with the exception of compress, where code quality is more important than in the other benchmarks). All the JIT compilers are below the baseline, because they provide at least level-0 optimization for all methods, even those executed only once. The AOS and AOS + FDO dynamic compilation methods perform best, giving a speedup of about 1.75, because they only take the time to perform compilation beyond baseline, possibly followed by additional optimizations, where there is good likelihood of performance benefit.

Figure 6.26 shows results for the steady-state benchmarks. Here, the results are quite different than for the start-up case. Now, the higher the optimization level for the JIT compilers, the better the performance. Using level 0 as a JIT gives a speedup of 1.9 over the baseline, and going up to level 2 yields a speedup of about 2.25. Interestingly, using the AOS staged optimization typically falls slightly short of the level-2 JIT. Because the same code is executed during each iteration, the JIT performance runs can complete compilation during the first iteration and then execute four iterations with no profiling or compilation overhead. On the other hand, the AOS performance runs profile and recompile through all five iterations. For two benchmarks, AOS without FDO actually beats JIT2, however, because AOS sometimes recompiles a method at a later time (in program execution) than JIT2 does. This later compilation allows more class loading to take place, eliminating some unresolved references, and thus the code generated by AOS is more highly optimized. Finally, because the AOS + FDO is able to perform additional profile-guided optimizations, for some benchmarks it provides significant speedup over the method-granularity optimizations used by the AOS implementation.

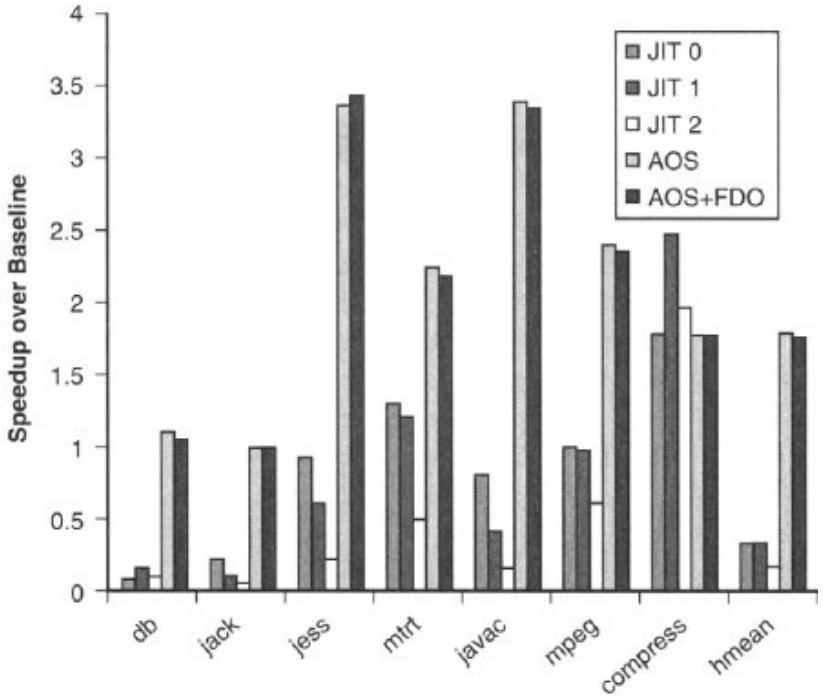


Figure 6.25 Jikes Start-up Performance on Seven SPECjvm98 Benchmarks.

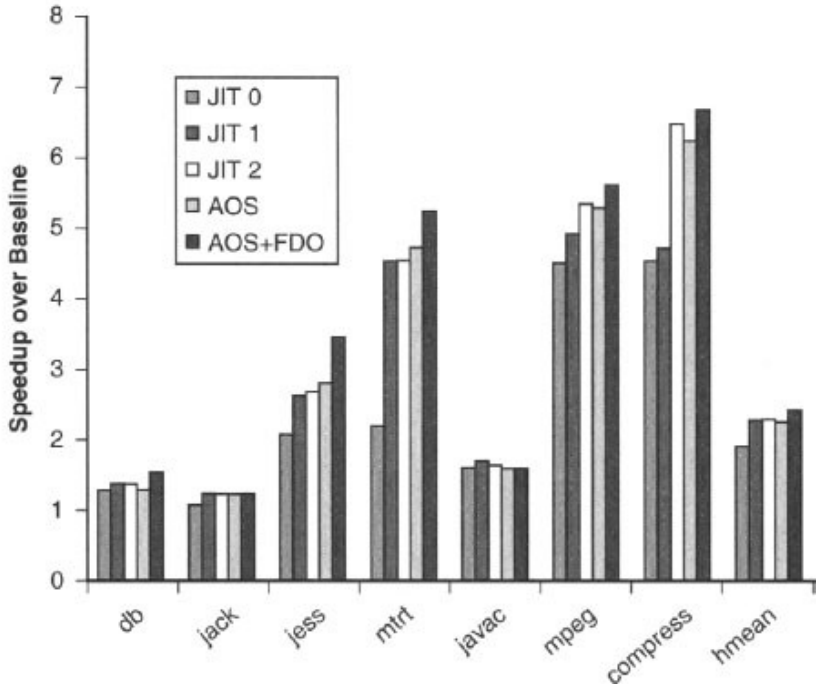


Figure 6.26 Jikes Steady-State Performance on Seven SPECjvm98 Benchmarks.

It is also interesting to see where an advanced JVM spends its time. An execution profile for Jikes is given in **Figure 6.27**. This particular

profile is for the entire SPECjvm98 benchmark suite, with each being executed for one iteration. Across different applications, the execution profile may vary significantly from this average performance, especially in time spent in garbage collection. Over the entire benchmark suite, performing optimized recompilation consumes only 7% of the total time, and all the other tasks related to adaptive optimization, i.e., the various control and bookkeeping functions, consume negligible time. We see that 86% of the time is spent in execution application threads and 6% is spent in the garbage collector. This tends to highlight the importance of garbage collection in a JVM; about as much time is spent in garbage collection as in performing optimizations on the code.

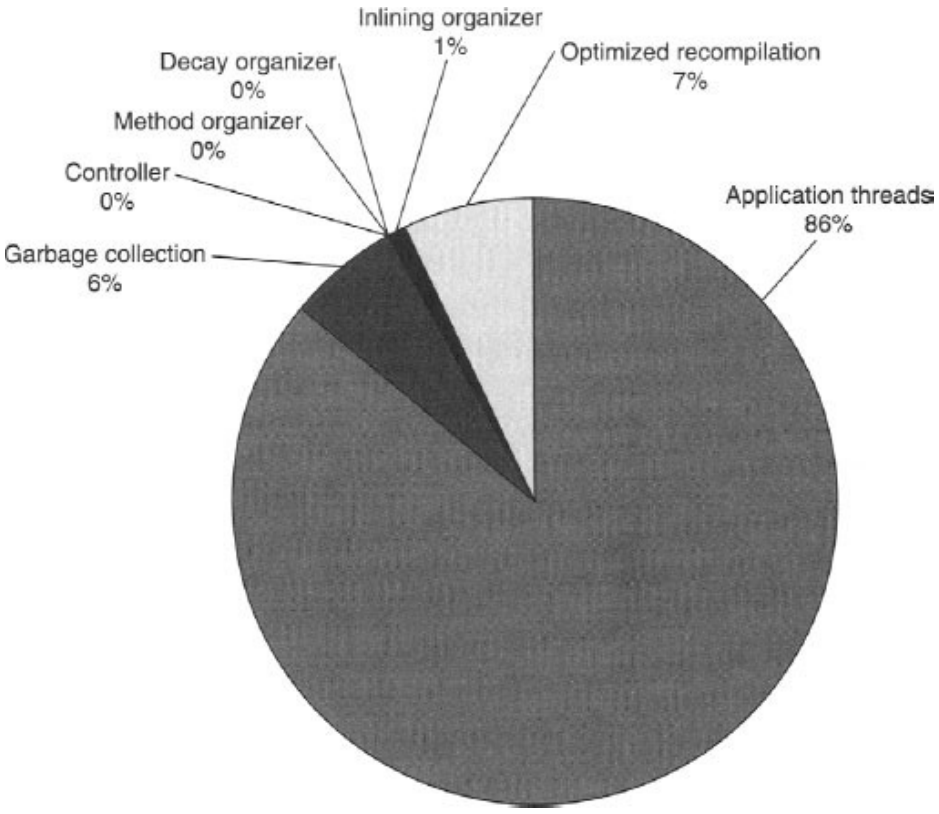


Figure 6.27 Where the Jikes System Spends Its Time.

6.8 Summary

The conventional belief is that programs running on an HLL VM are

relatively slow. This is a misleading notion. Some of the early Java VMs *were* slow. A heavy reliance on interpretation is naturally slow during steady-state emulation, and dynamic optimizing methods that are not selective, i.e., simple JIT compilation, will have high start-up times. However, modern HLL VM implementations rely on compiled code and are much more selective regarding when to apply optimizations and where. This leads not only to less overhead but to better optimizations in the steady state. Furthermore, slowdown due to the dynamic VM paradigm should not be confused with slowdown due to support for object-oriented programming. Object-oriented programming leads to more robust programs and a high level of code reusability (among other advantages), but sometimes there is a performance cost for these important software engineering advantages. Inherent in object-oriented code are additional levels of indirection that may not be found in conventional C-type code, for example. But this is usually caused by the object-oriented nature of the software, not the fact that it is running on a VM.

With the development of the Java platform and the more recent Microsoft .NET framework, HLL VMs already have achieved a "critical mass." At some point in the (not-too-distant) future, virtually all application programs are likely to be developed for a platform-independent HLL VM. This will change common programming practice (programming in C will be viewed in much the same way as programming in assembly language is viewed today), and it will likely change the way that hardware platforms are implemented. If all software is developed for a platform-independent HLL VM, the level of standardization is raised and the underlying hardware ISA becomes much less relevant.

---

1. It is not clear, however, that this type of behavior happens often enough in real programs to make such a technique worthwhile. This type of behavior is sometimes observed in kernel benchmarks (also known as *microbenchmarks*), and this may in fact be the prime motivation for considering such an optimization method in the

fact be the prime motivation for considering such an optimization method in the competitive benchmark-oriented environment in which high-performance VMs are often developed.

2. These yield points and profiling heuristics reflect the current Jikes implementation, not the one in the paper describing Jikes (Jalapeno) (**Arnold et al. 2000**).

Previous chapter
Chapter 5: High-Level Language Virtual Machine Architecture

Next chapter
Chapter 7: Codesigned Virtual Machines

Chapter Seven

Codesigned Virtual Machines

Both hardware and software have undergone radical changes since the first commercially available computers appeared in the early 1950s. In contrast, the basic character of the interface between hardware and software has been virtually unchanged during that time. Consequently, the instruction set architectures (ISAs) currently in use reflect a perspective and a division of labor between hardware and software that is decades old. At a time when hardware resources were very expensive and hardware was relatively simple, it made sense for the ISA to be a direct reflection of the specific hardware implementation. If the hardware had an accumulator, then the ISA had an accumulator. Over the years, however, hardware resources have become plentiful and inexpensive. Today, hundreds of millions of transistors are available on a single chip. As a consequence, underlying processor hardware has grown to be quite different from the image presented by the commonly used ISAs. For example, consider the Intel IA-32, one of the older microprocessor ISAs in use today. The IA-32 uses a CISC ISA, originally conceived for sequential in-order execution with relatively few general-purpose registers and stack-based floating instructions. A modern IA-32 implementation, however, converts the CISC ISA into what is essentially a RISC ISA running on a dynamic superscalar microarchitecture having lots of registers and register-based floating-point instructions (**Hinton et al. 2001**; **Keltcher et al. 2003**). This CISC-to-RISC conversion is done entirely in hardware (see **Figure 7.1**).

The reason behind hardware-intensive dynamic translation of legacy ISAs is the need to maintain their role as the interface between hardware and software. Today's ISAs are valuable interfaces

tween hardware and software. Today's ISAs are valuable interfaces, ones in which huge software and infrastructure investments have been made. Compatibility has also become the largest obstacle to implementing new ISAs that are better suited to today's technology. Fortunately, virtual machine technologies permit new ISAs by enabling a different approach to general-purpose processor design. Virtual machine technologies can turn processor development into a *codesign* effort, where the host architecture (the target ISA in particular) is designed concurrently with the VM software that runs on it. Because the host hardware is developed as an integral part of the VM, it is not necessary to use an existing host platform or target ISA. Taken together, however, the codesigned hardware and software support a conventional source ISA (and all the software developed for it).

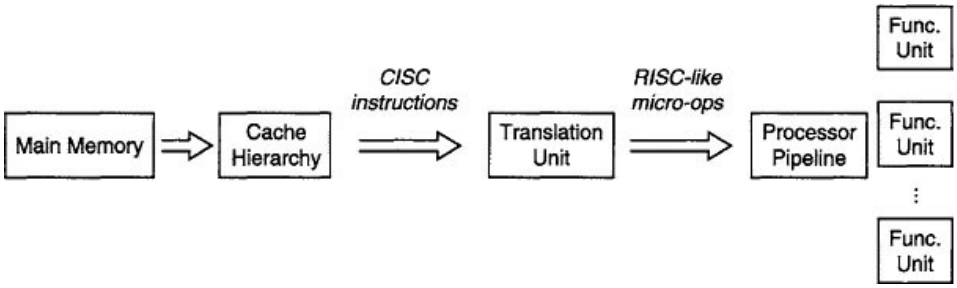


Figure 7.1 CISC-to-RISC Conversion. *Conventional high-performance microarchitectures for a CISC instruction set use hardware to translate CISC instructions into RISC-like micro-ops.*

These *codesigned virtual machines* open new avenues for architectural innovation. Because software becomes part of the "hardware" platform, the interface between hardware and conventional software is shifted upward, and there are new opportunities for dividing the implementation between hardware and software in an optimal way. Because codesigned VMs support an entire system, OS plus applications, they are a form of system virtual machine. However, unlike most other system virtual machines, codesigned VMs are not intended to virtualize hardware resources other than the processor, nor are they intended to support multiple VM environments. Rather, the goals include performance, power efficiency, and design

simplicity.

Perhaps the first codesigned VM was the IBM System/38 (**Bertsis 1980**), later known as the AS/400 (**Soltis 1996**) and currently the iSeries. The System/38 was designed to support a guest ISA with higher-level semantics than could be directly supported by hardware. This reduces the "semantic gap" between a conventional ISA and higher-level software (**Myers 1982**). A second objective was to separate the source ISA from the target ISA so that successive hardware platforms could be redesigned while maintaining software compatibility. The success of this approach was demonstrated as the AS/400 family evolved, changing its host platform ISA from the original proprietary CISC ISA to an extended PowerPC ISA. This migration took place in a completely transparent way from the users' perspective.

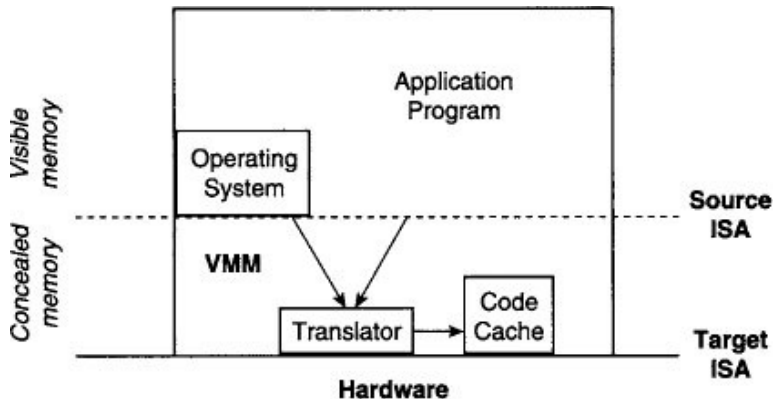


Figure 7.2 Virtual Machine Monitor. *A codesigned virtual machine architecture supports a conventional source ISA via a combination of hardware and software running in concealed memory.*

Another commercially available codesigned VM, developed by Transmeta (**Halfhill 2000**; **Klaiber 2000**) implements a target instruction set where a number of independent instructions placed in a very long instruction word (VLIW). The codesigned software is responsible for finding independent instructions in a conventional source ISA (the IA-32) and packing them into the target VLIW ISA. Because they are independent, the instructions in a VLIW can be issued for execution in parallel without complicated hardware sup-

port. Therefore, the advantage of this codesigned VM implementation is that it does not require the complexity (and power consumption) of an out-of-order issue unit as used in most superscalar processors.

An overview of a codesigned VM is shown in **Figure 7.2**. It is a system VM, as noted earlier, and all the VM software resides in a region of memory that is completely concealed from all conventional software. In keeping with the naming convention used for system VMs, we refer to the virtual machine software as a virtual machine monitor (VMM).

The primary function of the VMM is to emulate the source ISA; in this respect there are many similarities to process VMs. For example, emulation is often done in stages, with the focus on dynamic translation into a code cache. There are differences with respect to process VMs, however, the two most significant being the following.

1. There must be intrinsic compatibility (**Section 3.2**) at the ISA level rather than the ABI level. Because the interface at which virtualization takes place is the ISA, not only must the user-level instruction set be emulated, but the entire system-level ISA must be emulated. This includes the memory architecture, for example, so page fault compatibility must be maintained. It also means that I/O can be dealt with at the hardware level, not the abstracted OS call level as in process VMs.
2. The reason for using a virtual machine implementation is improved performance, power efficiency, design simplicity, or some combination of these. Compatibility is a requirement but is not the motivation for constructing a codesigned VM.

Codesigned VMs also bear some similarity to conventional superscalar processor designs (**Figure 7.1**): Both perform translation from a source ISA to a target ISA implemented in hardware. The difference is that one does the translation in software while the other does it in hardware. We should note, however, there have been a number of proposals for innovative processor implementations that are similar to codesigned VMs, typically relying on microcode or special-purpose coprocessors, rather than VM software to perform

translation or to assist with interpretation (**Debaere and Van Campenhout 1990**; **Nair and Hopkins 1997**; **Chou and Shen 2000**; Patel and Lumetta 2001). Nevertheless, in a conventional superscalar design of a CISC ISA, hardware (in one form or another) must carry the burden of performing the complex decomposition from CISC instructions to the RISC micro-ops that are directly executed. This is no doubt a contributor to the high cost of hardware design verification—a design cast into silicon is hard to debug and expensive to change.

Relying on hardware-based translation also limits the extent of the ISA translation that can be done. For example, inter-instruction optimization is difficult, and in a conventional superscalar implementation there is none. This is illustrated in **Figure 7.3**. With conventional hardware translation, **Figure 7.3**a, each source instruction is translated to micro-ops independent of the other source instructions, i.e., in a context-free manner. On the other hand, with software translation and optimization, as in **Figure 7.3**b, blocks of source instructions can be translated and optimized as a group, leading to more optimization opportunities (**May 1987**). Moreover, with the conventional hardware method, power consumption tends to be higher, due to the large amount of hardware for transforming source instructions into an executable form and scheduling them for execution.

Codesigned VMs are not as widely used as other types of VMs; there are few examples that are commercially available, and they remain the subject of research. They are of interest primarily because of their potential for supporting significant innovations in processor design. Furthermore, many of the hardware-based techniques developed for codesigned VMs can potentially be incorporated as extensions to existing ISAs so that other types of VMs, e.g., process VMs and HLL VMs, can be implemented more efficiently on standard platforms.

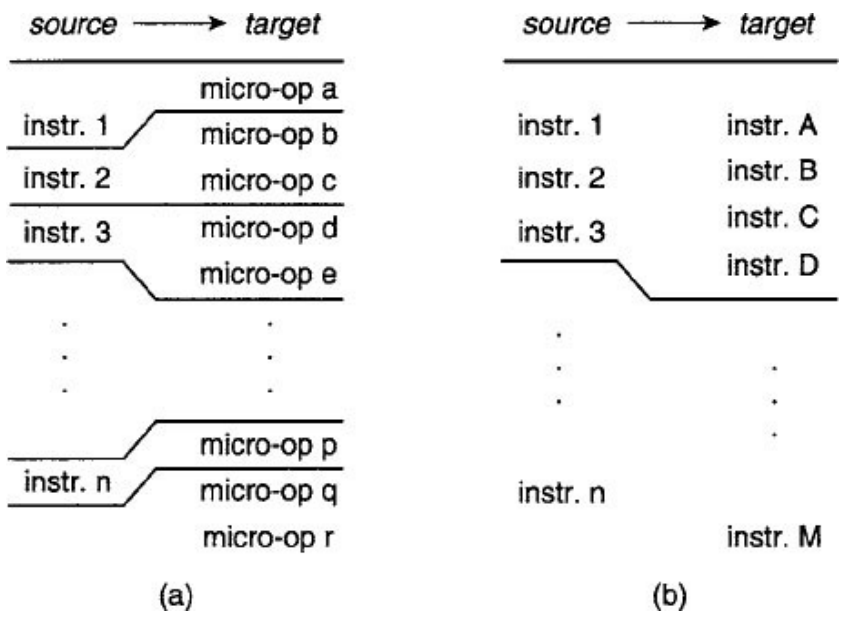


Figure 7.3 Code Translation Methods. *(a) Context-free translation, where each individual source instruction is translated to target instruction(s) (micro-ops); (b) context-sensitive translation, where blocks of source instructions are translated to blocks of source instructions.*

In the remainder of this chapter, we describe the implementation of codesigned VMs. This includes features that make the emulation process in codesigned VMs more efficient as well as specific performance and/or efficiency features that are enabled by the codesigned VM paradigm. The bulk of the chapter focuses on codesigned VMs intended to implement conventional ISAs; e.g., the Transmeta Crusoe implements the IA-32 ISA. It is this type of codesigned VM that has received the most attention in recent years, especially from researchers. Consequently, many of the techniques described in this chapter are centered on fast emulation of conventional ISAs. The IBM AS/400, in contrast, uses a source ISA that is at a higher level than most conventional ISAs and therefore relies more on innovative software than innovative hardware techniques to achieve its goals. The techniques used in the AS/400 are discussed primarily as part of the case study in **Section 7.8**.

7.1 Memory and Register State Mapping

State mapping is easier with a codesigned VM than with many conventional VMs. Because the target ISA is designed specifically for the source ISA, the host register file(s) can be made large enough to accommodate the guest's requirements, with extra scratch registers left over to enhance performance and/or simplify the translation process. An example of PowerPC state mapping for integer registers, as used in the IBM Daisy processor (**Ebcioglu et al. 2001**), is illustrated in **Figure 7.4**. Here, the PowerPC integer registers r0–r31 are mapped directly to host registers R0–R31. Then the branch unit counter and link register are mapped to registers R32 and R33. The PowerPC MQ register was originally used for multiplication and division but is now obsolete; in the original Daisy system it was mapped to host register R34. The constant value of 0 is placed in R35 to allow fast emulation of those cases where r0 is defined to be 0 in the PowerPC ISA. Finally, host registers R36–R63 are scratch registers to be used by the VMM emulator for such things as holding speculative values produced during code optimization, for constants, and for pointers to VMM tables.

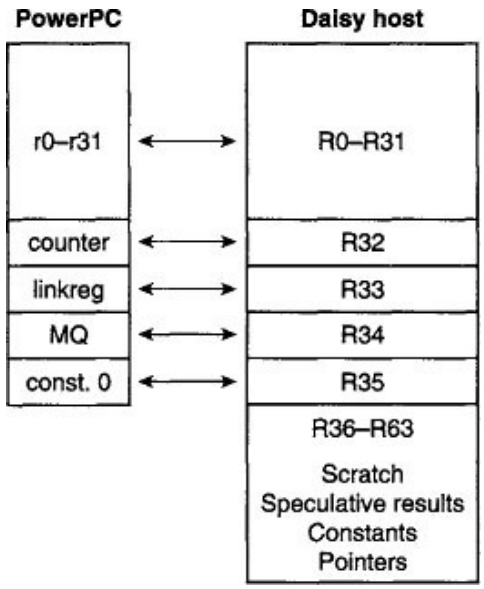


Figure 7.4 Mapping of Guest PowerPC Integer Registers to the Host Platform Register Set, as Done in the IBM Daisy System.

For memory state mapping, the key element of a codesigned system VM is the *concealed memory*. Concealed memory is a reserved region of the memory space where the VMM, code cache, and other emulation software and tables reside. There can be both concealed logical (virtual) memory and concealed real memory. The concealed memory is never made visible to any of the conventional guest software.

Because the VMM takes control immediately after system reset, it essentially has control of the system from the beginning, including the boot process, so it can make sure the conventional software never sees the concealed memory. When the guest OS boots (under VMM control) and checks to see how much real memory is available, e.g., by reading a control register, the VMM intercepts the read and does not inform it of the existence of the concealed real memory. And any attempt to access concealed memory by conventional software results in exactly the same behavior as if the memory were not present (typically a trap).

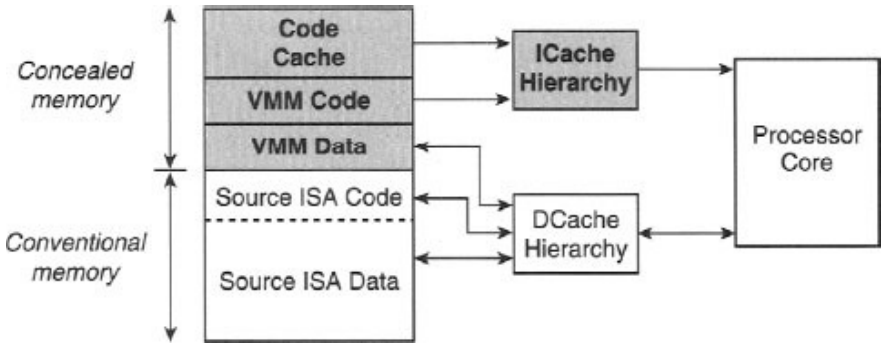


Figure 7.5 Memory System in a Codesigned Virtual Machine. *The shaded areas are those where host instructions and VMM data reside.*

Figure 7.5 illustrates an implementation of real memory in a codesigned VM. The shaded area of main memory is concealed, accessible only via the VMM. The shaded blocks hold target ISA instructions and VMM tables. The concealed memory region is of fixed size and does not change after system initialization. Instructions either are always translated and fetched from the code cache or are VMM

code; consequently the instruction cache hierarchy holds only target ISA instructions, and their presence in the instruction cache is concealed from the guest software.

With respect to logical addressing of memory, it is simplest to allow the guest OS to manage the conventional (nonconcealed) part of real memory as it normally would and to map guest virtual addresses using the guest ISA's address translation architecture. For mapping and addressing of concealed memory, there are a number of options, some of which are illustrated in **Figure 7.6**.

In the first option, concealed logical memory shares an address space with the guest (**Figure 7.6**a). The conventional memory addressing and address translation are maintained as in the guest system, so the host address space must be enlarged, with the guest space fitting inside. Because the guest space is likely to be consistent with one of the "standard" sizes (e.g., 32 or 64 bits), the host space is either the next-larger "standard" size (e.g., 64 or 128 bits) or a nonstandard size (e.g., 33 or 65 bits). This approach is straightforward, but the expansion of the logical address space size may make the host ISA awkward or expensive to implement (which defeats the purpose of a codesigned VM).

A second option is to have two separate logical address spaces, one for concealed memory and one for conventional memory (**Figure 7.6**b). All the VMM software, code cache, associated tables, etc. are placed in the concealed address space, which maps to concealed real memory. This approach implies that load and store instructions must select one of the two mapping tables (e.g., by using different opcodes to select the mapping table or by using a mode bit). Note that all instruction fetches are to concealed memory addresses. Because the VMM controls all instructions that are executed (either through interpretation or because VMM-translated code is executing), the VMM can always ensure that the proper load and store instructions are used.

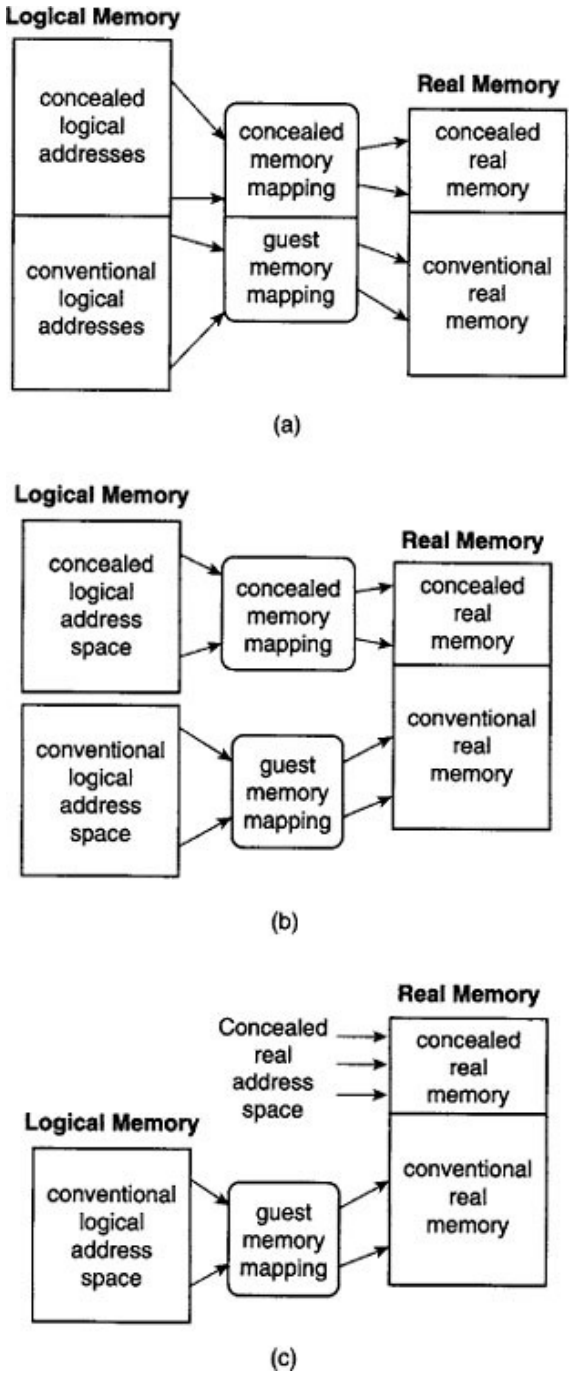


Figure 7.6 Methods for Mapping Concealed Memory. *(a) Extend the host address space; (b) provide a separate address map; (c) use real addresses.*

A third option is to use real addressing for concealed memory, i.e., to bypass address translation when accessing concealed memory (**Figure 7.6**c). In a sense, this is a special-case form of option 2. One way of implementing this is to provide separate sets of load/store in-

structions for accessing memory, one with real addresses and the other via the guest mapping table(s). Also note that the VMM has the option of bypassing address translation even when accessing conventional real memory. In effect, the VMM always operates within the real address space and uses logical addressing only when emulating the guest software. With this approach, a mode bit can then be used for selecting the type of memory addressing to be used, rather than using opcode-based selection.

A key issue regarding concealed memory is how far into the memory hierarchy the concealed memory reaches. It is easiest to keep the concealed portion of memory in RAM and not allow it to extend to secondary storage (disks). If the concealed memory extends to the disk, then the disk (and any bus addresses it uses) must also be concealed from the guest OS. Furthermore, the VMM becomes responsible for managing the concealed secondary memory, including any paging that it might support. Because VMM and the code cache can be kept relatively small (on the order of a few tens of megabytes) and because it simplifies the system design, most of the proposed and implemented codesigned VMs use the diskless option, with VMM software being stored in ROM. The extension of concealed memory to secondary storage is discussed in **Section 7.5**.

7.2 Self-Modifying and Self-Referencing Code

In a codesigned VM, self-modifying and self-referencing code are generally handled using the techniques described in **Section 3.4.2**. As pointed out in the previous section, it is easiest to keep the original guest OS's virtual-to-real page mapping intact. Then, because the source code is held in the guest's memory in its original form, any load or store accesses to an instruction page will naturally proceed correctly.

For self-modifying code, any attempt to write into the guest code region must be caught, and this is most easily done by write-protecting the source code region, so that any attempt to write it will cause a trap that can be directed to the VMM. The VMM can then flush the code cache of translations derived from the modified page and allow the page to be written.

In a typical process VM (as in **Chapter 3**), the VMM can write-protect a page via a system call to the host OS in order to change the page table protections. In a codesigned VM, however, the solution is a little more complicated because it is the guest OS that manages its own page tables, not the VMM. This includes the pages that contain the page tables themselves, so changing the page protections in a guest page table would violate the principle of keeping all of the guest memory state intact. The most straightforward solution is to use the TLB to enforce write-protection of guest code pages. If the TLB is managed by the VMM, as it normally would be, then the TLB entries are enhanced with a special "write-protect" bit, for use only by the VMM for detecting self-modifying code. Whenever an entry for a code page is loaded into the TLB, the VMM sets the write-protect bit. In order to keep track of the source code pages, the VMM maintains a table of all the guest virtual pages from which it has translated (or interpreted) code. This is done at the time code is translated or interpreted.

To reduce performance losses due to self-modifying code (both real and pseudo-self-modifying), the software-based methods described in **Section 3.4.2** can be applied. In addition, because of the codesigned paradigm, special hardware support can be provided to further reduce performance losses caused by pseudo-self-modifying code. In the Transmeta Crusoe, a special hardware structure is added to speedup fine-grained write-protection checking (**Dehnert et al. 2003**; **Banning et al. 2002**). The special hardware structure is managed by the VMM and is similar to a small software-managed

TLB. This *write-protect table* can hold the fine-grained write-protect masks (**Section 3.4.2**) for a few source code pages, along with hardware for comparing a potentially faulting store address with the bits in the bit mask (see **Figure 7.7**). After an address translation is performed with the normal TLB, the real address is presented to the write-protect table. If the normal TLB detects a write-protect fault, the smaller write-protect table can automatically filter out those cases where the write is not to a translated code region. As noted earlier, the TLB write-protect bit is a special bit used solely for protecting code pages. The VMM loads the write-protect table with the bit masks for a set of source code pages that are actively being written; this is usually a small number, if there are any at all.

A final issue involves I/O writes to guest code memory. If an I/O device writes to a guest code page, then these writes must also be caught. Again, the VMM helps enforce this by keeping track of all the real guest pages for which active translations exist in the code cache. To do this, the VMM has to maintain a hardware table for I/O writes, most easily held in the memory controller. The VMM makes entries in this table for all the real pages that hold guest code pages. A store to any of these pages results in an interrupt to the VMM, which can then flush the translations that are derived from the guest code page.

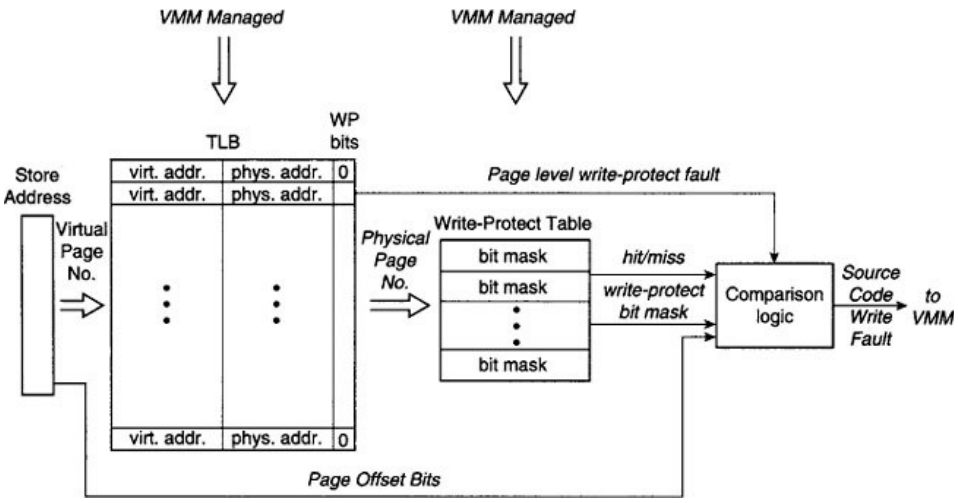


Figure 7.7 Fine-Grained Write-Protection Method for Source Code Regions. *A fault is*

Figure 7.7 Fine-Grained Write-Protection Method for Source Code Regions. *A fault is triggered to the VMM if the TLB indicates a write-protect fault and there is a miss in the write-protect table or the comparison logic indicates the write address is to a write-protected fine-grained code region.*

7.3 Support for Code Caching

A central element of a codesigned VM implementation, as with other emulating VMs, is the code cache. Because performance and efficiency are key factors in a codesigned VM, code cache performance is of utmost importance. It is useful to consider the issues that may lead to performance losses when using a code cache. Because the blocks of translated code held in a code cache are of unequal size, accessing the code cache becomes more complex than accessing a conventional cache, where blocks are of equal size. A code cache access begins by hashing the source PC (SPC) value to an entry in a map table, reading a corresponding SPC value (or tag) from the map table, and performing a comparison to determine if there is a hit (**Figure 2.28**). If so, the target PC (TPC) at the map table entry can be used for accessing the code cache. If the first probe of the map table mismatches, then additional probes of the table may be required in order to handle potential hash collisions.

The access delay for the code cache can be quite long, due to the multiple memory accesses and indirect jumps required. A single dispatch table lookup can take on the order of 10–15 instructions, leading to significant performance losses. In a simple code cache implementation, all control-transfer instructions (branches and jumps) perform a map table lookup. However, to improve performance for direct jumps and branches, superblock chaining (**Figure 2.29**) is an effective solution. With this method, the map table lookup for direct jumps and branches is eliminated entirely. For indirect control transfers the problem is more difficult, however, because register-indirect jumps have their destination addresses stored in a register, and destination addresses can change during the program's execution. Furthermore, the register address is the SPC value, not the TPC

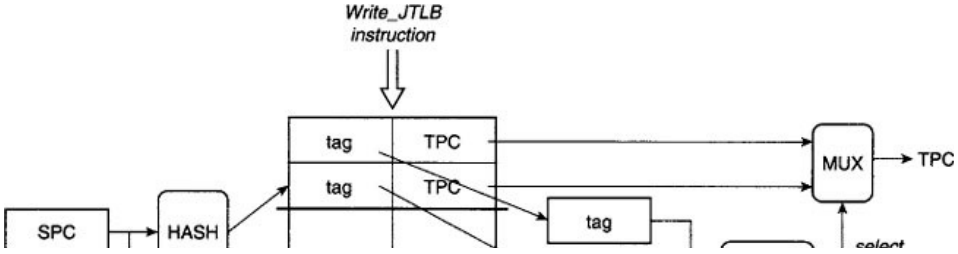
value. This means that the source jump address held in a register must be translated from SPC to TPC every time the translated indirect-jump instruction is executed. A straightforward approach is to consult the map table for every indirect jump; but, as noted earlier, this is slow and hence indirect jumps will add a significant performance cost.

To save table lookup overhead for each and every indirect jump, many dynamic optimizers/translators implement a form of software-based jump target prediction (see **Section 2.7.2**, illustrated again in **Figure 7.8**). Here, a sequence of instructions compares the indirect SPC address held in a register against an embedded translation-time SPC address. A match indicates a correct "prediction," and the inlined direct branch instruction to a TPC is executed; if there is no match, then the code jumps to the slow map table code.

The software prediction method is of somewhat limited value, however. First, if the software prediction is incorrect, then time is wasted by first testing the possibilities when the dispatch table lookup has to be performed anyway. Second, there are a number of indirect jumps that are very difficult to predict using this method, for example, returns for procedures that have a number of call sites and therefore a number of constantly changing destination addresses. The indirect-jump problem is probably the greatest source of performance loss in a software-only code cache system.

```
if ((Rx) == #addr_1) goto #target_1
else if ((Rx) == #addr_2) goto #target_2
else map_lookup (Rx)        ; do it the slow way
```

Figure 7.8 Example of Software Indirect Jump Prediction.



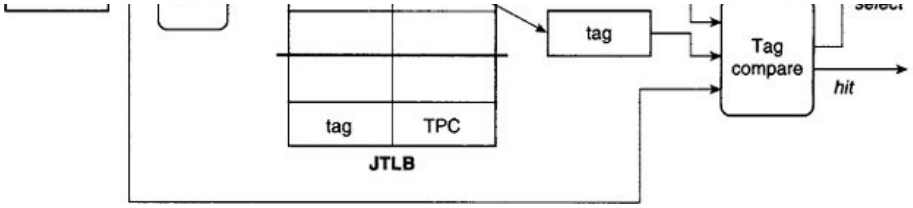


Figure 7.9 Jump Translation Lookaside Buffer (JTLB). *This example uses a two-way set associative table.*

7.3.1 Jump Translation Lookaside Buffers

In a codesigned VM, the high-overhead software map table can be replaced with a specially designed hardware cache of map table entries (**Gschwind 1998a**; **Kim and Smith 2003**). We refer to such a table as a *jump translation lookaside buffer* (JTLB), because it is similar in concept to the software-managed address TLBs used in virtual memory systems. A JTLB is illustrated in **Figure 7.9**. Each entry in the JTLB contains a tag and a TPC value. The JTLB can be fully associative, set associative, or direct mapped. This example is a two-way associative table. Entries are written into the JTLB by the VMM and always contain correct translations (this can be ensured, for example, by requiring that if a translated block is removed from the code cache, then the VMM must remove any associated entries from the JTLB). The JTLB is accessed by hashing the SPC value (this can consist of taking upper address bits as in a conventional TLB or cache, or it can be a more complex hash function). After the hash, the tag value is compared with the full or partial SPC, depending on the hash function used. If there is a JTLB hit, then the TPC value stored in the entry that hits is the desired jump address in the code cache.

A JTLB can be integrated into a codesigned ISA in a couple of ways. The first is a JTLB_Lookup instruction that accesses the JTLB with an SPC address held in a register and then reads out a TPC and places it in a second register. A third register (or condition code) indicates a hit or miss in the JTLB. An example is in **Figure 7.10**. Here, the JTLB_Lookup instruction takes an SPC value in Rk, puts the associated TPC value in Ri (if there is a hit), and places the hit/miss out-

come in Rj. This instruction is followed by a conditional jump on the value in Rj; if there is a miss, control transfers to the map_lookup routine. This routine will place the correct code cache mapping in the JTLB (if the code has been translated); otherwise, the VMM will begin interpretation or will translate the accessed code. A second, simpler way to integrate the JTLB into the ISA is to combine the lookup with the conditional jump, i.e., a Lookup_Jump Rk instruction that performs the jump to the TPC if there is a hit, otherwise it falls through.

```
JTLB_Lookup Ri, Rj, Rk      ; TPC to Ri, hit/miss to Rj
Jump Ri, Rj==0              ; conditional indirect jump
Jump map_lookup             ; do it the slow way
```

Figure 7.10 An Instruction Sequence That Accesses the JTLB.

This method still requires that when a jump is encountered, instruction fetching must stall until the JTLB is accessed and the jump is executed. An additional enhancement is to predict the TPC value immediately after a Lookup_Jump instruction is fetched, using a more or less conventional branch target buffer (BTB) (see **Figure 7.11**). Then the predicted instruction stream is accessed from the code cache immediately. Meanwhile, the Lookup_Jump proceeds up the pipeline (along with the predicted TPC). When the Lookup_Jump instruction issues and accesses the JTLB, the prediction is checked. A misprediction causes a pipeline flush, with fetching being redirected to the correct TPC. A miss in the JTLB causes a flush and a fall-through in the original code sequence (which would then jump to the map_lookup routine).

7.3.2 Dual-Address Return Address Stack

Either the software prediction approach or the JTLB approach, when supplemented with a BTB prediction, can be very effective, as long as the BTB prediction is correct a high percentage of the time. For many indirect jumps this will be the case, but for procedure return jumps it is often not the case. The problem is that a procedure

may be called from a number of places, so a return jump can have a number of different targets that frequently change.

To deal with this problem in a conventional microarchitecture, most modern processors employ a hardware return address stack (RAS) mechanism that can predict a return instruction's target address very accurately (**Kaeli and Emma 1991**). It basically mimics the software procedure stack by pushing the fall-through PC onto a hardware prediction stack whenever there is a procedure-call (jump) instruction. However, in a codesigned VM, this approach cannot be used in its conventional form because the saved return destination address would be an SPC, whereas the corresponding TPC is actually needed for a return destination address prediction. Furthermore, if the procedure jump is at the end of a translated superblock, as it often would be, then the address of the instruction following the jump is not the correct return address anyway (the correct address is at the beginning of a different superblock).

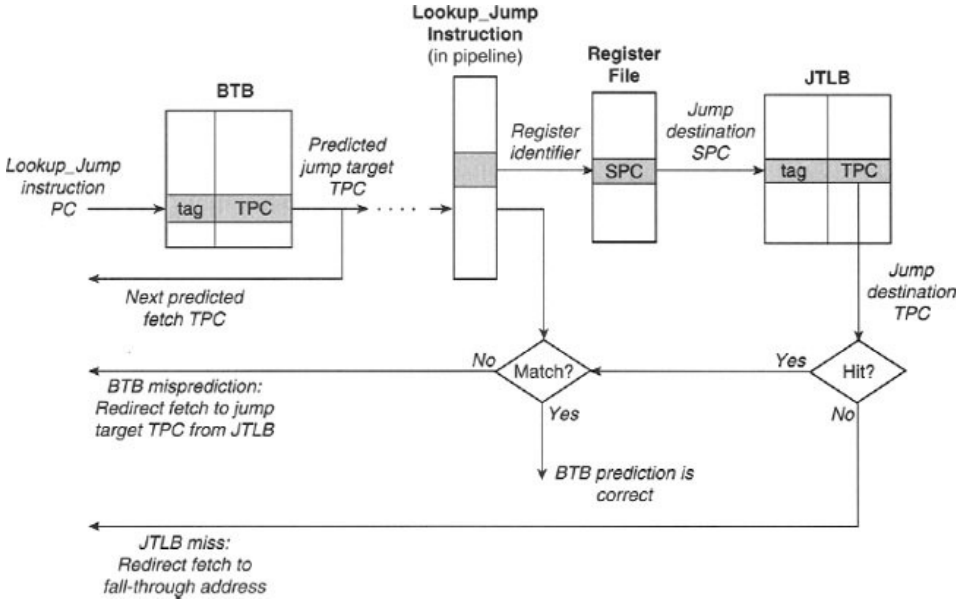


Figure 7.11 An Enhanced Lookup_Jump Instruction That Uses a BTB to Predict the TPC.

In a codesigned VM, a specialized dual-address RAS mechanism can be used for return address prediction (**Gschwind 1998b**; **Kim and Smith 2003**). Each entry in the dual-address RAS (DRAS) contains an

Smith 2003). Each entry in the dual-address RAS (DRAS) contains an address pair, consisting of a return address SPC and its corresponding TPC, as shown in Figure 7.12. The DRAS can be thought of as a hardware implementation of the shadow stack mechanism used in the FX!32 (see Section 2.7.3).

In order to push an address pair onto the DRAS, a special push-DRAS instruction pushes both return addresses. Finding the return destination SPC at superblock construction time is straightforward. Finding the TPC value is similar to setting up superblock linking. If the corresponding TPC is not known at superblock construction time, an invalid address is written in the TPC field of the push-DRAS instruction. Later when the return target superblock is constructed, the invalid address is replaced with a valid TPC. Alternatively, if a JTLB is present, it is accessed at the time of a procedure call, and the TPC value is pushed onto the prediction RAS. In either case, the prediction RAS should be pushed/popped for all procedure calls and returns—sometimes even with an invalid TPC to maintain correct stack ordering.

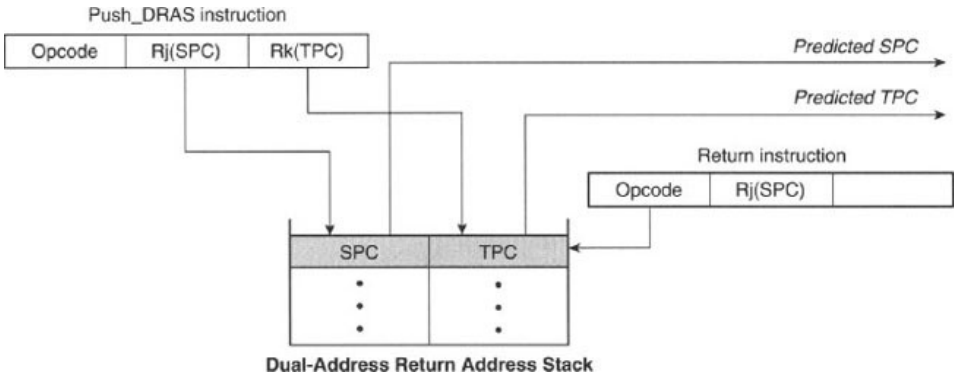


Figure 7.12 Dual-Address Return Address Stack. *The Push_DRAS instruction pushes both SPC and TPC. The return instruction uses TPC as a prediction.*

When a return instruction is fetched, the next fetch address is predicted with the popped TPC. The SPC part of the pair flows down the pipeline with the return instruction and is compared to the register value when the return instruction is issued. If the two values do not match, a RAS misprediction is detected, and fetch needs to be redi-

rected. Fetch redirection is accomplished by a jump to the dispatch table code. This is done by having the return jump instruction "fall-through" to the next instruction if there is a misprediction. The next instruction is a jump to the dispatch table code. Note that if a JTLB is also used, a simplification is to replace the *dual* RAS with a *single* RAS, where only the TPC is pushed. The JTLB can be relied upon to provide the correct TPC.

7.4 Implementing Precise Traps

Just as in other VMs, implementing precise traps is an important problem in codesigned VMs. In general, one can use techniques similar to those described in **Chapters 3** and **4**. That is, during the translation process, the optimizing software maintains software checkpoints. If code motion is involved, register live ranges are extended to ensure that checkpoint values can be restored when there is a trap. Then interpretation beginning at a checkpoint prior to a trapping instruction establishes the correct, precise state. In codesigned VMs this approach is facilitated because the host ISA can be designed to have enough registers so that live ranges can be extended without excessive register pressure. Nevertheless, this software-based approach does limit the types of code motion that can be performed (see **Section 4.5.1**). For example, many instructions cannot be moved below store instructions. By adding hardware support for checkpoints, this restriction on code motion can be removed, thereby relaxing the requirements for register live range extension.

7.4.1 Hardware Support for Checkpoints

In codesigned VMs, the designer has the option of providing hardware support for precise traps in translated code. The basis for the approach is to use hardware to set a checkpoint at the time each translation block is entered (**Figure 7.13**a). Then if there is a trap anywhere in the translation block, the state at the beginning of the

block can be restored via the hardware. At that point, interpretation is used to provide the precise exception state, just as with the software method. The basic sequence is illustrated in Figure 7.13b.

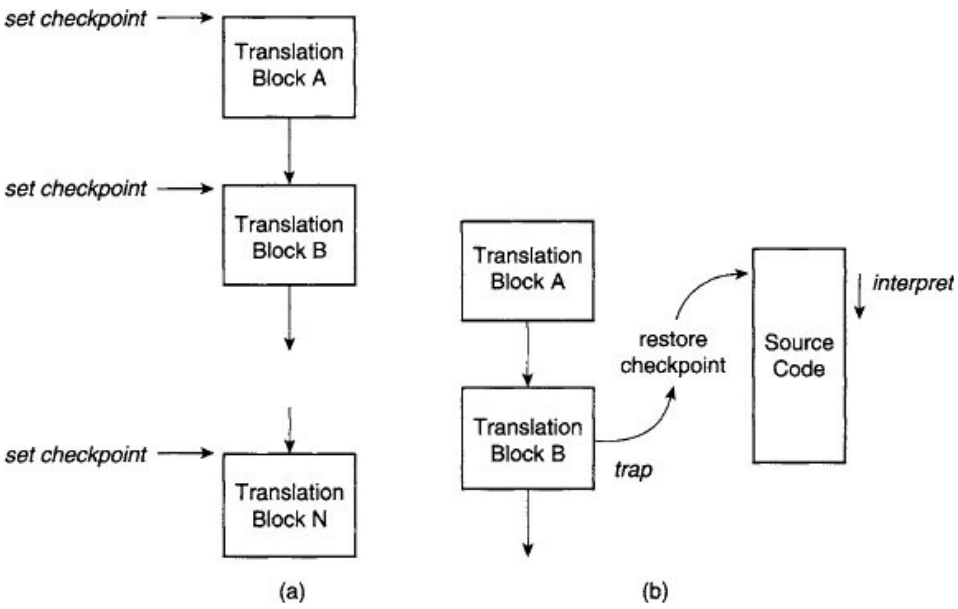


Figure 7.13 Hardware-Supported Checkpointing. *(a) Checkpoints are set at every translation block entry point. (b) When there is a trap, the checkpoint is restored, and interpretation begins at the beginning of the source code that formed the trapping translation block.*

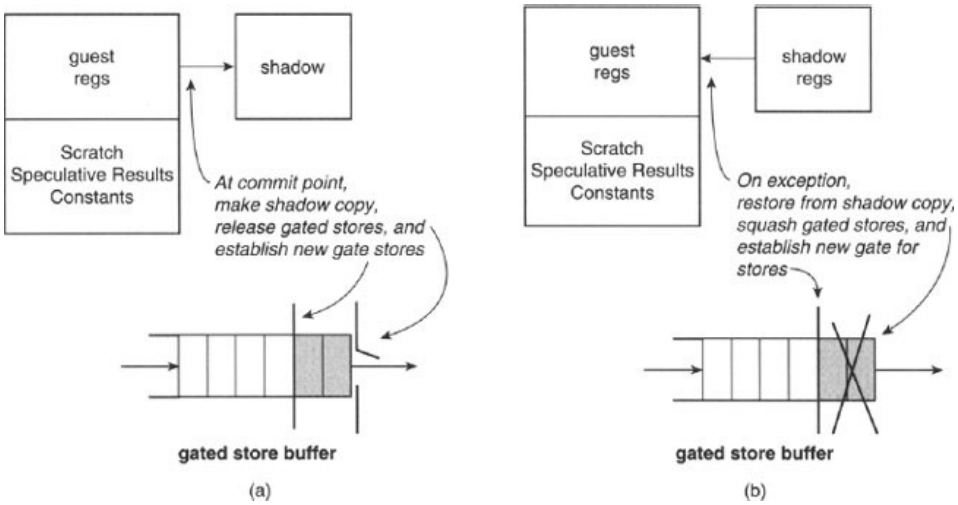


Figure 7.14 Checkpoint Hardware Operation. *(a) When a checkpoint is committed (and a new one is established), the gated store buffer is released and a new checkpoint is established. (b) If a trap is detected, the contents of the store buffer are flushed and the shadow copy is written back into the register file.*

The mechanism for supporting checkpoints is illustrated in Figure 7.14 (Klaiber 2000). When a new translation block is entered, the state from the previous block is committed, and a new checkpoint is

state from the previous block is committed, and a new checkpoint is set (**Figure 7.14**a). Setting register checkpoints is fairly easy. The host hardware can support a shadow copy of the guest registers (**Smith, Lam, and Horowitz 1990**), and registers are checkpointed by copying them *en masse* into the shadow registers. With proper hardware design (and circuit layout) this copying can be done in a very small number of cycles (as few as one). An alternative approach uses shadow registers at a finer granularity (**Nystrom et al. 2001**) and avoids discarding all the computation done in a code block when an exception is detected.

Checkpointing memory in a similar manner would be much harder because keeping a shadow copy of main memory would be very expensive, in both time and space. Hence, an alternative is to keep all memory store operations buffered in the processor until a translation block is exited (and its state changes can be committed) (**Klaiber 2000**). Refer to **Figure 7.14**a. At the time a checkpoint is committed and another is set, a gate is closed at the tail of the store buffer so that any instructions following the gate are inhibited from completing. Meanwhile, the gate at the head of the store buffer (established at the previous checkpoint) is opened so that stores can complete by writing to memory. These writes can occur in the background as computation proceeds. Note that all load instructions must check the store buffer as well as the data cache, in case a load is to an address that is in the buffer. This includes preceding, uncommitted stores in the same translation block and any residual, committed stores from a preceding block.

If an exception should occur during execution of the translation block (**Figure 7.14**b), then the buffered stores are flushed and the registers are restored from the shadow copy. At that point the state is the same as it was when the block was first entered. Then interpretation of the source code can take over to finish the job of locating the trapping instruction and providing a precise state.

An advantage of the hardware-supported checkpoint method is that the code inside a translation block can be reordered by software in any fashion (subject to true data dependences, of course), even when the block includes store instructions. Hence, the restrictions on code motion given in **Section 4.5.1** are relaxed. The only limitation is the fixed size of the store buffer, and this can constrain the translation block size in some cases (the number of stores in the block cannot exceed the store buffer size).

On the other hand, software reordering of loads and stores can lead to violation of memory ordering compatibility. In a shared-memory multiprocessor there are hazards associated with reordering loads and stores as observed by other processors (see Appendix Section A.7.3), which means that reordering may be limited in shared-memory multiprocessor implementations. The reason is that after the software reorders the instructions, the original order is lost; yet correct operation in parallel programs may depend on the original ordering. A method for dealing with multiprocessor memory ordering is described in a patent by members of the IBM Daisy project (**Altman et al. 2002**). This method is based on acquiring exclusive access to cache lines as stores enter the store buffer and then flushing the buffer and restarting if a different processor should attempt access to the line before the stores in the buffer are released.

7.4.2 Page Fault Compatibility

Thus far, we have not considered the recovery of the precise guest state when page faults occur. In most process VMs, page faults are not an issue because they are manifestations of OS policies, are handled by the OS, and are hidden from the process. A codesigned VM is a system VM, not a process VM, which means that the guest OS must observe exactly the same page faults as it would if it were running on a native platform. If the guest OS has mapped a logical page to a conventional real page, then it must not trigger a page fault; similarly, if the guest OS has not mapped a page and it is accessed, then

it must page fault. Implementing page fault compatibility is relatively simple for the data region of the guest's memory. If the host implements the same memory mapping as the guest (**Figure 7.6**) and lets the guest OS manage conventional memory, then page faults to the data region will be detected naturally by the host system. These faults will return control first to the VMM, which can use techniques as described in **Section 4.5.2** to determine if the fault is spurious (due to optimizations and/or code reordering) and to produce the correct, precise state before control is handed over to the guest OS.

Likewise, when interpreting instructions, the interpreter loads instructions from the conventional region of memory, so page faults for instruction fetches will occur naturally. For executing translated instructions from the code cache, however, the problem is more difficult because the guest instructions are not actually fetched; rather, translated instructions are fetched from concealed memory. The codesigned VM must therefore implement some mechanism that will trigger a page fault when a translated instruction is fetched from the code cache and the corresponding guest instruction *would have* caused a page fault had it been fetched on a native source platform. There are a number of possible approaches to solving this problem. One approach is *active,* another is *lazy.* Both approaches are discussed in the following paragraphs. Note that in this discussion we assume an architected page table and page faults. Special considerations for an architected TLB and TLB faults will be discussed in a separate subsection.

Active Page Fault Detection

The active approach to instruction page fault detection is to monitor potential page replacements by the guest operating system. When a source instruction page is replaced by the OS, all the translation blocks based on the replaced source page should be flushed from the code cache.

the code cache.

To determine when an instruction page is potentially being replaced by the guest OS, the VMM monitors the guest OS's modifications to the architected page table. This can be done by marking memory that holds the page table as "write-protected." Assuming the page table is architected, the region of memory that holds the page table can be identified by the VMM. Write-protecting pages is most easily done if VMM software manages the TLB. Then the write-protection information is added to the implementation of the TLB and does not require any changes to the architected page table in main memory. The VMM should also monitor any other operations that may change a virtual to real page mapping, e.g., changes to the page table pointer.

In addition, the VMM keeps a table of virtual page numbers for each page containing source instructions. At the time a block of source instructions is translated, an entry is made in the table (if it is not already in the table). Then whenever the page table mapping is modified, there is a trap (or jump) to the VMM, and the VMM determines if the page table entry for a source instruction page is being modified. If so, the VMM flushes all the translations in the code cache derived from that page. This requires a pair of side tables. One side table tracks all the translation blocks derived from each page; these are the translation blocks that must be flushed. The second table (or an extension to the first), keeps track of any link backpointers (see **Section 3.8.2**). The link pointers are changed so that they point into the VMM emulation manager. After this is done, an attempt to transfer control into the removed page will instead jump to the emulation manager. Subsequently, the emulation process will detect the instruction page fault when it attempts to emulate an instruction from the code page and it misses in the page table. Finally, if hardware structures containing SPC-to-TPC address translations are implemented, such as the JTLB and DRAS (**Section 7.3**), then these structures should be flushed of any entries corresponding to flushed

translations.

Lazy Page Fault Detection

With the lazy solution, when a source code page is replaced by the guest OS, the code cache is not immediately flushed of corresponding translations. Rather, it waits until there is an attempt to actually use the translated code. To accomplish this, every time the translated code crosses a source page boundary, the page table is probed to see if the mapping is the same as on the original translation (**Ebcioglu et al. 2001**).

The method is illustrated in **Figure 7.15**. First, it must be possible to determine when translated code crosses a source page boundary. This is most easily done if all the code in a translation block is derived from the same source page. If this is the case, then source pages boundaries are crossed only when there are link jumps, e.g., when block HIJ jumps to block KL in **Figure 7.15**. Otherwise, if translated code can come from different source pages, then the crossing points within the translated code should be distinct, at least with respect to precise state updates (the transition from E to F in **Figure 7.15**). In either case, at the time a source page boundary is crossed, the first instruction is a Verify_Translation instruction, inserted by the translator. This instruction probes the page table to see if the page mapping has changed. The Verify_Translation instruction provides the virtual address of the newly entered source page along with the real address at the time the block was translated. The instruction attempts to translate the virtual page number to a real page number and, if successful, compares it with the real address provided by the instruction. If the virtual page is mapped to real memory and the two real addresses match, then execution of the translated code proceeds. Otherwise, the VMM takes over and generates a page fault and/or translates the source code page that has just been entered.

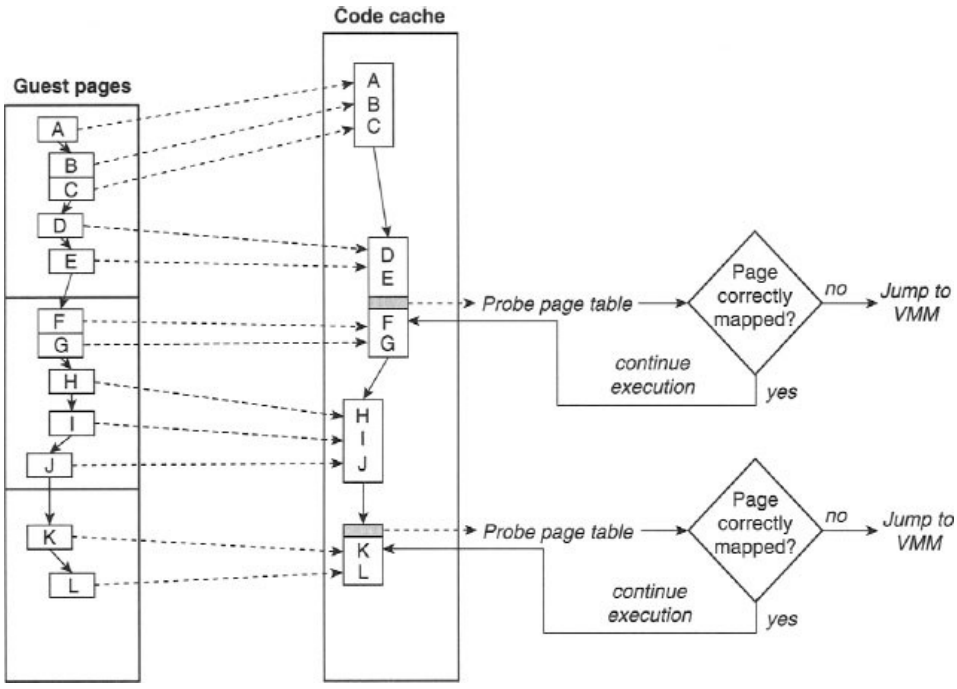


Figure 7.15 Detecting Instruction Page Faults via Instructions That Probe the Page Table.

Architected Translation Lookaside Buffers

With architected TLBs, both the active and lazy methods can be adapted in a straightforward way. But, because of performance issues, the active approach is somewhat problematic. If the active approach is used, then translations are flushed every time a TLB entry for a source code page is overwritten. Then the code must be retranslated if the TLB entry is later restored. If there is a fairly large number of source instruction pages and a lot of TLB activity for instruction pages, this approach could lead to a very high overhead. With the lazy approach, Verify_Translation instructions can be used, as before. The only difference is that they generate an architected TLB fault rather than a page fault when the translation is not present.

7.5 Input/Output

For the most part, implementing I/O in a codesigned VM is straight-

forward. If the VMM does not use any I/O devices itself, then all the guest device drivers can be run as is (although instructions in the driver code are emulated, just as all other guest instructions are). Any I/O instructions or memory-mapped I/O is simply "passed through" by the VMM so that the I/O system sees the same signals that it would if the guest were running on a native platform.

As noted in **Section 4.5.1**, memory-mapped I/O results in volatile memory locations, and the presence of volatile memory locations should inhibit optimizations involving removal and/or reordering of memory loads and stores to those locations. To identify code regions that access volatile memory, an access-protect bit, similar to the write-protect bit of **Section 7.2** can be added to the TLB. Then if a memory-mapped I/O routine performs loads or stores to an access-protected memory page, there is a trap to the VMM, which can then deoptimize translations containing the accesses to volatile memory in order to ensure that they all occur in the correct sequence (**Kelly, Cmelik, and Wing 1998**). As an additional enhancement, the implementation ISA can be extended to include special volatile versions of load and store instructions, which do not trap when the access-protect bit is set; if these are used in deoptimized I/O code, then their execution will proceed without trapping.

If a VMM uses no I/O devices, then it is restricted to reside entirely in concealed memory. This means, for example, that code translations cannot be cached to disk and reused as is done in the FX!32 system described in **Section 3.10**. All translations must be done from scratch each time a program is run; and when translated code is removed from the code cache, it is discarded and must be retranslated before it can be used again.

If the requirement of complete transparency can be relaxed somewhat, then concealed memory can be extended to secondary storage (disk). In order for this to be accomplished, there must be a portion of a disk (or possibly an entire disk) reserved for use by the VMM as

concealed secondary storage. The most straightforward way to do this is to provide a special disk driver that is VMM-aware (**Figure 7.16**). It is the loading of this special disk driver that breaks complete VMM transparency. This limits the use of the codesigned processor to those operating systems for which the special disk driver has been developed and deployed. The disk driver code can restrict the OS to that portion of the disk that is visible (actually, the guest OS is only informed of the nonconcealed region). Meanwhile, the VMM uses the disk driver to access its concealed region of disk.

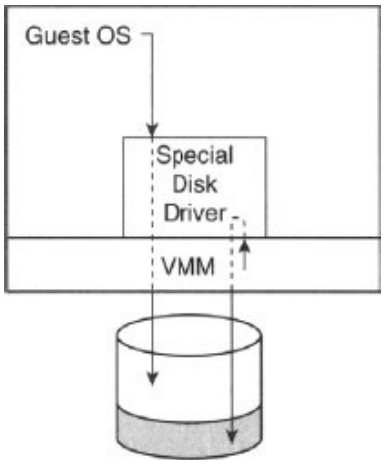


Figure 7.16 Concealed Secondary Storage. By adding a special disk driver, concealed secondary storage (shaded) can be implemented. Both the guest OS and VMM software can call the driver, but only the VMM can access the concealed region of the disk.

To avoid relaxing transparency even more, the VMM is entirely responsible for managing the concealed secondary memory. This implies that it is not backed up or archived, for example, so this restricts its use to caching code translations (or for paging VMM software). This large, persistent code cache would be useful for reducing initial start-up time for big programs (including the OS when it is booted). However, checks have to incorporated to make sure that any source code with cached translations has not changed since the time of the translation, possibly by keeping a copy of the original source code and performing an instruction-by-instruction check (**Conte, Sathaye, and Banerjia 1996**). Furthermore, the concealed secondary storage would only be useful for large blocks of translated codes—for small blocks it would probably be faster to trans-

lated codes—for small blocks it would probably be faster to translate from scratch than to make disk accesses. An interesting (and less transparent) disk-based code caching approach was proposed for dynamic rescheduling of VLIW binaries based on page-sized code blocks (**Conte and Sathaye 1995**).

7.6 Applying Codesigned Virtual Machines

Thus far, we have described a number of mechanisms that allow a codesigned VM to emulate full ISAs and execute instructions efficiently from a code cache. These mechanisms are only enablers; however, to be useful these mechanisms must be applied in codesigned VMs that provide real advantages, such as improved performance, power efficiency, and software flexibility. Advantages can be achieved at two levels, in effect at a macro level and at a micro level.

At the macro level, entirely new ISAs can be implemented. Some of the proposed and implemented target ISAs for codesigned VMs have been aimed at exposing instruction-level parallelism to hardware in a more efficient manner than with conventional instruction sets. Most conventional instruction sets still reflect the sequential model of execution prevalent decades ago. Consequently, hardware is burdened with discovering parallelism in the instruction stream at run time; this is the essence of modern superscalar processors.

Two of the better-known codesigned VMs, the Transmeta Crusoe (**Klaiber 2000**; **Halfhill 2000**) and the IBM Daisy/BOA (**Ebcioglu et al. 2001**; **Sathaye et al. 1999**), implement VLIW instruction sets as their target ISA. The codesigned software is responsible for finding independent instructions in the source binary and packing them together in the VLIWs. The advantage of a VLIW implementation is that it does not require the complexity (and power consumption) of an out-of-order issue unit as in most superscalar processors.

A proposal for a different style of codesigned VM has a similar goal —to simplify instruction issue logic—but it does this by exposing chains of dependent instructions to the hardware (rather than independent ones as with a VLIW) (**Kim and Smith 2003**). Then multiple in-order instruction issue units can be given the dependent chains of instructions. Each issue unit processes instructions in order, which simplifies hardware, but the multiple issue units can issue instructions out of order with respect to each other. A research project at the University of Illinois (**Merten et al. 2001**) provides a number of interesting codesigned hardware components targeted primarily at profiling and optimization, rather than ISA innovation.

The goal of the IBM AS/400 implementation (**Soltis 1996**) is to provide a high-level object-oriented source ISA. Besides being well suited for efficient support of object-oriented system and application software, this approach allows many of the hardware resource management mechanisms, e.g., page management, to be placed in the implementation-dependent VMM. Another advantage of the codesigned approach was demonstrated by the AS/400 designers when they successfully replaced the original host ISA (a proprietary CISC) with an extended PowerPC ISA and migrated users in a completely transparent way.

At the micro level, codesigned VMs permit the implementation of specific performance enhancements (which may be related to the selected host ISA). For example, in a VLIW platform, all code scheduling is performed by the software translation system, so the ability to reorder instructions is crucial to the success of a VLIW-based computer. Consequently, in the codesigned VMs that use VLIW target ISAs, there is special support to enable reordering of instructions, especially load and store instructions, because dependences among load and store instructions are difficult to determine statically (i.e., to *disambiguate* them).

Finally, an important feature of codesigned VMs is that implementation-dependent profiling hardware can be built in for use by dynamic translating/optimizing software (**Conte, Menezes, and Hirsch 1996**; **Heil and Smith 2000**; **Merten et al. 2000**). This profiling hardware can be matched to optimizations used by the codesigned software translation code as well as features in the microarchitecture.

To make the foregoing advantages more concrete, the next two sections give case studies of two of the more important codesigned VMs, the Transmeta Crusoe and the IBM AS/400.

7.7 Case Study: Transmeta Crusoe

Although some elements of the codesigned VM paradigm had been used previously, for example, to provide code portability, the Transmeta Crusoe broke new ground in using a codesigned VM for achieving power efficiency and design simplicity. The TM5000 series was introduced in early 2000; these processors are described here. The follow-on Efficeon (TM8000) series was introduced in 2003, it incorporates larger caches than the TM5000 series, and its VLIW instructions are twice as wide.

The Crusoe implements the Intel IA-32 ISA with an underlying proprietary VLIW instruction set that uses dynamic code translation and optimization, centered on a code cache. The microarchitecture of the TM5800 processor is illustrated in **Figure 7.17**. The VLIW (or a *molecule,* in Transmeta parlance) consists of four instructions (*atoms*). It is a classic VLIW in the sense that the individual instruction fields in the VLIW are dedicated to specific functions. These instruction fields feed a branch unit, a floating-point unit, an integer unit, and a load/store unit. For precise traps, the Crusoe uses the method described in **Section 7.4** and is based on shadow registers and a gated store buffer.

and a gated store buffer.

The memory hierarchy is illustrated in **Figure 7.18**. The VMM software is held in compressed form in a 512KB boot ROM. At the beginning of the boot process, this software is decompressed into a 2MB region of concealed memory. In the implementation shown, the code cache and various side tables consume 14MB. There are conventional 64KB level-1 instruction and data caches, but, in addition, there are two smaller memories dedicated to the VMM. These are for critical VMM code and data that the designer relies upon for low-latency access.

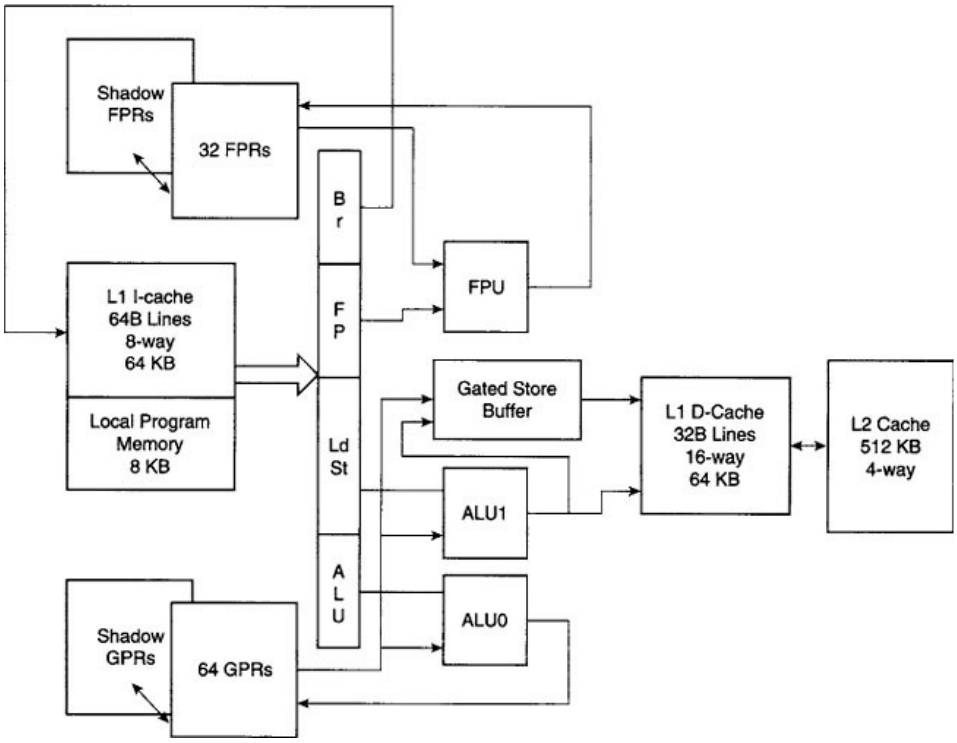


Figure 7.17 Transmeta Crusoe TM5800 Microarchitecture.

The big optimization challenges in the Crusoe involve breaking the complex IA-32 instructions into RISC-like micro-ops and then finding parallelism and scheduling them into VLIWs. Because an in-order VLIW microarchitecture is used, code reordering is a key part of the optimization process. To provide support for reordering, the Crusoe designers built some special features into the implementation ISA.

tion isA.

Being able to move load instructions is perhaps the most important type of reordering, because load instructions take longer to execute than most other instructions, especially if there is a cache miss. That is, it is desirable to reorder a load instruction higher up in the instruction stream so that it executes earlier and can therefore eliminate or reduce the wait time of subsequent dependent instructions. For example, consider the code in **Figure 7.19**a, which contains two loads and two stores with an add instruction that consumes the two loaded values. It would be desirable to move the load instructions higher in the code sequence, as is shown in **Figure 7.19**b. Then there are other intervening instructions that can overlap with the execution time of the loads. Consequently, the time the add instruction might have to wait for its input operands is reduced, especially if one or both of the load instructions should miss in the data cache. However, moving the load instructions up may be unsafe, because the store

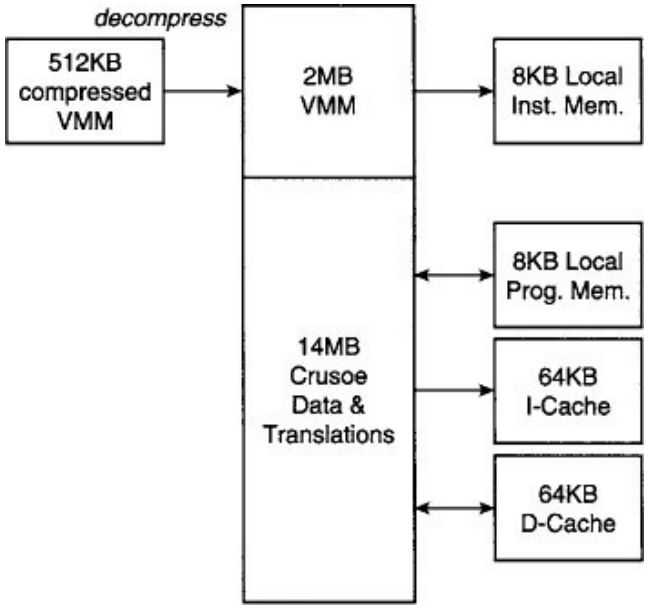


Figure 7.18 Transmeta Crusoe Memory Hierarchy.

Original code	*Rescheduled (unsafe)*	*Rescheduled (protected)*
`st 0(r1),r2`	`ld r3,0(r4)`	`ldp r3,0(r4) x`

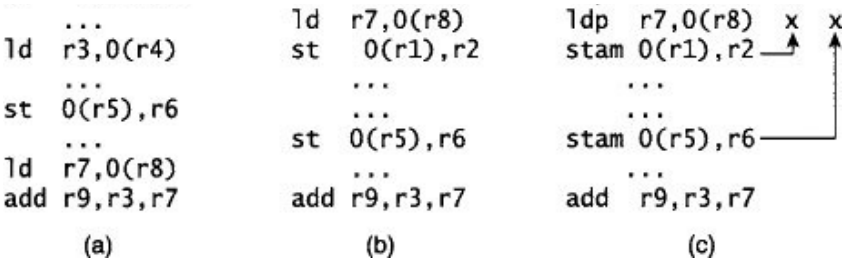


Figure 7.19 Safe Reordering of Memory Operations with Load-and-Protect and Store-under-Alias-Mask Instructions. *(a) Original code sequence; (b) unsafe reordering (due to unknown addresses); (c) reordering with protection from 1dp and stam instructions.*

instructions may be to the same memory address. For example, if the store to 0(rl) happens to be to the same address as the load from 0(r4), then in the reordered code the load will return the wrong data. Because the register values are often not known until run time and they may change over the course of the program's execution, the binary translator is not able to guarantee that the load and store addresses are never the same, so, to be safe, such a code reordering should be inhibited.

A codesign approach for detecting such memory conflicts is very well suited to this problem (**Gallagher et al. 1994**), and the Transmeta designers took such an approach by adding a pair of special instructions. The load-and-protect instruction (1dp) performs a load operation and records the address and size of the loaded data in an architected table. The store-under-alias-mask (stam) instruction performs a store and also contains a mask that identifies certain table entries—the ones set by preceding 1dp instructions. For each stam instruction, the masked 1dp instructions are illustrated in **Figure 7.19**c with arrows from the stam to the 1dp; the table itself is not shown. The second stam in the example identifies only the second load, because the first load was already ordered ahead of it. If a table entry indicated by the mask of a stam instruction detects an overlap between the 1dp's memory address and the stam's memory address, then there is a trap to the VMM. At that point, the VMM can handle the situation like it does most other traps: It can go back to the previous checkpoint and begin interpreting the original code in order to get the correct result. If a particular translation block re-

order to get the correct result. If a particular translation block repeatedly traps, then the VMM can reorder the code more conservatively, taking care to leave the overlapping load and store in their original sequence.

The 1dp/stam pair is an excellent example that illustrates the interplay between the codesigned hardware and software in a codesigned VM. In this case, where a VLIW is used, the ability to reorder code is very important, and the 1dp/stam instructions, in conjunction with the dynamic optimization code, facilitate code reordering.

7.8 Case Study: IBM AS/400

The architecture of the IBM AS/400 (derived from the earlier System/38) is perhaps not as widely known as other "textbook" computer architectures, but it is both highly innovative and commercially successful (**Soltis 1996**). The AS/400 includes a codesigned VM—in fact, the whole system, including system software, was designed from scratch using a fully integrated approach, so one could say the entire system was codesigned from top to bottom. Unlike the other codesigned VMs discussed in this chapter, however, the System/38 was not designed to support an existing, conventional ISA. Rather, the original System/38 designers invented a new, high-level ISA with the goals of software simplicity and machine (hardware) independence.

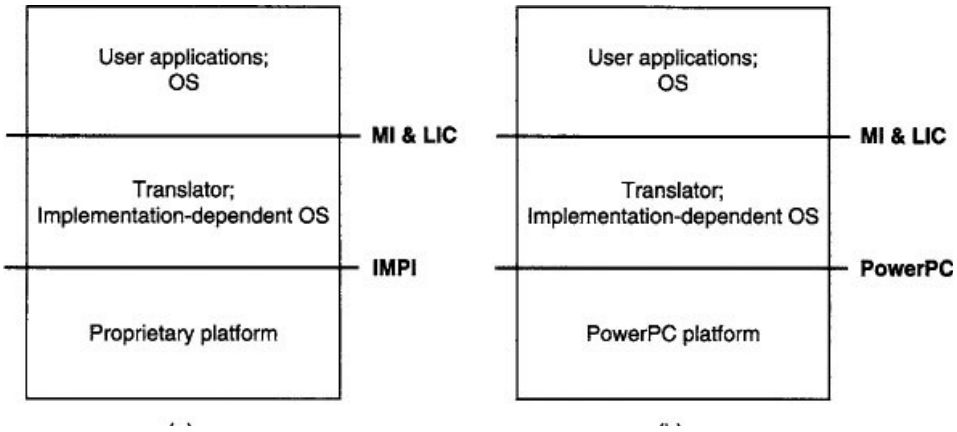




Figure 7.20 AS/400 Architectures. *(a) The original System/38 and early-generation AS/400 systems were based on a proprietary ISA (IMPI) and hardware platform. (b) More recent AS/400 systems are based on an extended PowerPC ISA and platform.*

The high-level instruction set is referred to as the *technology-independent machine interface,* or just MI for short. Supplementing the MI is a set of standard libraries, the Licensed Internal Code (LIC), which deals with the implementation-dependent aspects of resource management. The MI and LIC together from an ABI layer upon which all implementation-independent software runs.

In the original design, the underlying implementation architecture was based on a proprietary CISC ISA called the *internal microprogrammed interface* (IMPI)[1] (see **Figure 7.20**a). In developing a high-level ISA that is implementation independent, the objective was to provide for future changes in the underlying implementation, including the lower-level ISA. This feature has been exploited as the AS/400 evolved, first by extending the IMPI in a transparent way and then by a complete shift to an extended PowerPC ISA (**Figure 7.20**b).

Because the MI is at a higher level than a conventional ISA and was developed with the goal of hardware independence, the AS/400 architecture is similar in philosophy to the HLL VMs described in **Chapter 5** and follows the model illustrated in **Figure 5.1**. That is, the MI compiler is analogous to a conventional compiler frontend, and the MI translator is analogous to a conventional compiler backend. However, like other codesigned VMs, the AS/400 is a system VM. It supports not only language aspects of program execution but also a full system environment. The philosophy of dividing the implementation-independent and implementation-dependent software extends to the OS functions; for example, device drivers and memory management algorithms are in the implementation-dependent part of the OS. These functions are embodied in the licensed internal code, which uses the host ISA (IMPI or PowerPC).

The AS/400 uses an object-based ISA (the MI). So in this way it also follows a similar philosophy to the Java VM and the CLI (**Chapter 5**). However, in the AS/400, the MI itself is object based, in the sense that there are architected object types and type-specific instructions built into the MI. The programs that run above the MI do not necessarily have to be object based, however. In contrast, with Java and the CLI systems, the V-ISA includes architected primitives that are then capable of supporting the object-oriented programs that run above the V-ISA level; the only architected objects with their own instructions are arrays.

7.8.1 Memory Architecture

The MI has a memory architecture composed of objects. The objects are completely isolated from one another and can only be accessed via pointers; see **Figure 7.21**. There are a number of object types supported and used by MI instructions. The simplest type of object is simply called a *space,* which, as the name suggests, consists of a space for data. Other objects contain object-specific information, i.e., a "functional part" as well as space for constants and other object-related data. An object can be accessed only through a pointer created specifically for that object. To access a location inside a space, a space pointer or data pointer (for typed data) must be used. The functional part of an object is implementation dependent and is manipulated only by MI instructions specific to that type of object; the contents of the functional part are not otherwise visible above the MI. The MI-supported objects can be created, destroyed, and modified by programs.

Actual address values contained in pointers are not made visible to software above the MI level. Pointers can be used only for accessing data, and pointers cannot be modified with ordinary instructions. If an attempt is made to write to a memory location containing a pointer, then its ability to be used as a pointer is immediately de-