S/390 microprocessor design

by C. F. Webb

The technical, business, and market requirements for large enterprise servers ("mainframes") strongly influence the design of microprocessors for these systems. Specific characteristics of the ESA/390 and z/Architecture instruction set architectures lead to different pipeline and branch prediction strategies than are found in most other microprocessors. The requirements for robust, scalable performance across a wide range of workloads, highly efficient logical partitioning, and very high hardware reliability affect the cache structure, internal code, and hardware fault detection and recovery designs.

Introduction

At one level, all microprocessor designers face essentially the same challenges of balancing performance, schedule, and cost while conforming to the physical rules of the silicon technology and to the logical requirements of the instruction set architecture (ISA) for which the processor is being designed. This commonality of constraints tends to produce considerable similarity among microprocessor designs across the industry. At another level, however, unique requirements for the ISA, system applications, or a targeted market can drive a particular microprocessor into a somewhat different region of the design space.

Such is the case with S/390 microprocessor designs, which face particular challenges on three fronts:

- Compatibility with existing software.
- Optimization for high-end server applications.
- Reliability, availability, and serviceability (RAS).

These requirements are all closely associated with the S/390* "mainframe" system platform, and are essential to the overall value of S/390 in the marketplace. While S/390 processors strive to be competitive across a wide range of applications, and thus bear many of the characteristics of processors designed for other platforms, these requirements affect the design in a variety of ways.

Following a discussion of these market-based requirements for S/390, we consider how these requirements influence key design decisions in the instruction processing pipeline, the handling of branch operations, storage operand access, and cache organization. We then consider the design implications of complex instructions and special S/390 architectural facilities, and close with a brief discussion of hardware error detection and recovery in S/390 processors. Throughout, our focus will be on those factors in the market requirements and S/390 architectural definition which exert particular, and in some cases unique, influence on S/390 microprocessor design.

S/390 requirements

The most obvious unique requirement for S/390 is the ESA/390 architecture [1] and its 64-bit successor, z/Architecture [2]. These ISAs are direct descendants of the S/360 architecture introduced in 1964. As new implementations of this architecture have been introduced, and as the ISA itself has evolved from S/360 to S/370* to 370/XA to ESA/390 to z/Architecture, compatibility of new hardware with old software has been a persistent feature of the platform. New functions and features are added in

0018-8646/00/$5.00 © 2000 IBM

an upwardly compatible way so that customers need not modify or even recompile their programs to maintain existing function. Furthermore, performance advances in S/390 (and predecessor) processors have been optimized toward running existing code well, so that mainframe customers have been able to achieve steady growth in system performance without having to change their software when they change hardware.

This focus on upward compatibility is readily apparent in the definition of z/Architecture, which provides support for 64-bit real and virtual addressing and 64-bit operations without requiring any modification to existing ESA/390 applications, allowing those applications to coexist with new programs that fully exploit the new architectural features. This leads, for example, to the requirement that 32-bit ESA/390 operations leave unchanged the high-order 32 bits of the 64-bit general-purpose registers on which they operate, which in turn affects the implementation of those registers and operations in the z900 (64-bit successor to S/390) design. A further consequence of this focus on compatibility and on a smooth transition to 64-bit functionality is the requirement that programs achieve the same high level of performance whether they are written to a 32-bit ESA/390 model or fully exploit the 64-bit capabilities of z/Architecture.

Another aspect of the S/390 platform which drives the processor design is the focus on the use of S/390 in large configurations as a high-end server system. This usage results in a large number of distinct tasks being run in parallel on different processors, and in close time proximity on the same processor; it also leads to a high degree of concurrency of access to the same memory locations by multiple processors. On S/390, the PR/SM* hypervisor and the OS/390* operating system both exploit the strong memory coherence definition in ESA/390 to obtain maximum throughput from a multiprocessor S/390 system. To support this highly efficient manner of operation, an S/390 system design must provide high-bandwidth, low-latency, cache-coherent access to all parts of system memory, and must deal effectively with large working sets at the cache block, page translation, and address space levels. This has its greatest impact on the memory hierarchy and multiprocessor interconnection design [3, 4], but has implications for the microprocessor as well, particularly in the cache design and address-translation design.

A third distinct feature of S/390 is the emphasis on reliability, availability, and serviceability (RAS). For customers who rely on S/390 systems for their mission-critical, enterprise-wide operations, this characteristic is indispensable. System-level RAS is achieved via a rich combination of hardware, Licensed Internal Code (LIC), and system software, which together must detect, isolate, contain, and recover from a wide range of possible hardware and software errors, with minimal impact on customer operations [5].

Instruction set optimization

The ESA/390 ISA is a distinctively CISC (complex instruction set computer), as opposed to RISC (reduced instruction set computer) architecture. That is, it has variable-length (2, 4, and 6 bytes) instructions, register-to-storage and storage-to-storage operations, complex operations requiring LIC execution, and a large set of instructions. The relative merits of CISC and RISC approaches have been widely debated, but in recent years the distinctions have been blurred at the microprocessor level, as RISC design techniques have been applied to CISC designs. For S/390 the debate is moot, since efficient execution of programs (including OS/390) written to the existing ESA/390 ISA is a basic product requirement. This requires relatively complex instruction fetching and decoding logic to handle the different instruction lengths and the large instruction set, as well as some form of internal code to deal with operations too complex for hard-wired execution logic. Beyond these features, the instruction format and set of operations *per se* do not necessarily require a unique design for an S/390 microprocessor, since the basic functions required are essentially the same as for any other ISA.

On the other hand, achieving optimal performance on programs written to the ESA/390 ISA requires that the microprocessor design reflect the specifics of the ISA, yielding in some cases quite different design decisions than would be appropriate for a different ISA. Many such decisions are implicit in S/390 processor designs, and are reflected in the instruction pipelines and architectural structures they employ [6–9]. In many cases the differences between these designs and those of other microprocessors are readily apparent; less obvious is the connection between these differences and the ESA/390 ISA.

Branch and condition code operations

Consider the handling of condition registers and branch operations. In ESA/390 there is a single 2-bit condition register known as the condition code (CC), which is updated unconditionally by a large number of arithmetic and logical operations. In contrast to ISAs which contain multiple condition registers and which provide explicit program control over which condition register (if any) is set by any given instruction, the ESA/390 definition affords relatively little opportunity to separate the setting of the condition code from the testing of that code via a conditional branch, since only instructions which do not affect the condition code are allowed in this interval. In many RISC microprocessors, conditional branch instructions are isolated from other instructions very early in the instruction pipeline and are resolved (executed) immediately if the condition being tested has been computed. This minimizes the delay associated with an incorrect branch prediction, but it depends upon some degree of separation in the program between the instruction computing the condition (loading the condition register) and the conditional branch (using the condition register). If the condition-setting instruction immediately precedes the branch instruction, resolution of that branch will be delayed until the execution logic downstream in the pipeline can process that instruction and communicate the resulting condition information back to the branch logic. If this is a common situation, as it is with ESA/390 programs, the advantage of early execution of conditional branches may be lost.

An alternative design typically used in S/390 microprocessors is to resolve conditional branches later in the pipeline, in the same stage (i.e., with the same pipeline latency) as fixed-point instruction execution, using a special branch unit in some designs and the general or fixed-point execution logic in others. At this point in the pipeline, the condition code is generally available, even from the immediately preceding instruction. This avoids the upstream control flow in the instruction pipeline between condition code update and branch resolution, eliminating the load-use latency for the condition code and relieving the compiler or programmer of any need to separate the setting of the CC from the conditional branch. The downside of this design is that the penalty for branch misprediction is increased, because branch resolution (i.e., discovery of misprediction) is delayed until the execution portion of the pipeline. Note, however, that this penalty is incurred only for incorrectly predicted branches, the frequency of which is greatly reduced by the use of sophisticated branch prediction techniques (see below).

A corollary benefit to the preference of S/390 microprocessors for execution-time branch resolution is that a compiler for ESA/390 need not be concerned about scheduling instructions between the setting of the condition code and the corresponding conditional branch. As noted above, this is difficult in any case given the ESA/390 definition, but the design of IBM S/390 CMOS microprocessors makes it irrelevant for performance, since the branch incurs no additional delay by immediately following the condition code update. Removing this constraint frees a compiler to focus on other code optimizations, some of which are more important for S/390 than for other platforms.

Another distinctive feature of ESA/390 is its heavy reliance on register-based branch target addresses. This follows from the general form of S/390 operand addressing, which uses a base general register, an offset from that base, and in some cases an index general register. Although relative branch instructions, in which the branch target is specified as an offset from the current instruction address, have been added to ESA/390, these are not yet in widespread use, and the vast majority of the branches executed on S/390 computers use a general register to specify the target address. (These are often referred to as indirect branches.) A consequence of this is that branch target-address generation requires access to the general register file and must handle all of the usual pipeline-register interlocks. This may be contrasted with a design for an ISA in which indirect branches are the exception rather than the rule; in the latter case, the branch target address can generally be computed by adding the specified offset to the current instruction address, with no register file access required and no interlock against prior instructions.

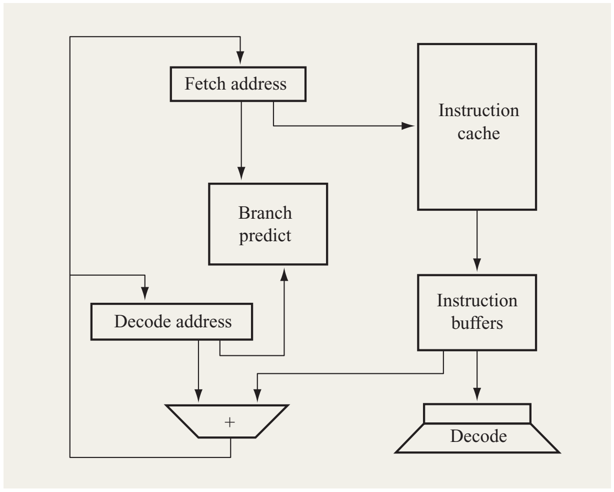

Figure 1

Branch prediction optimized for relative branches.

Where relative branches are dominant, dedicated hardware is generally added to compute the branch target immediately upon recognizing the branch instruction, in some cases doing so prior to the normal instruction-decode stage of the pipeline. When combined with a relatively small and low-latency instruction cache, this allows the target stream for a taken branch to be fetched and made ready for decoding with little delay. This lessens the need to predict branch target addresses ahead of time, and is reflected in designs that emphasize elaborate schemes for predicting branch *direction* but spend little or no hardware predicting branch *targets*, or which confine target prediction to indirect branches. This approach is illustrated in **Figure 1**, with the direction-prediction mechanism(s) being accessed at fetch and/or instruction

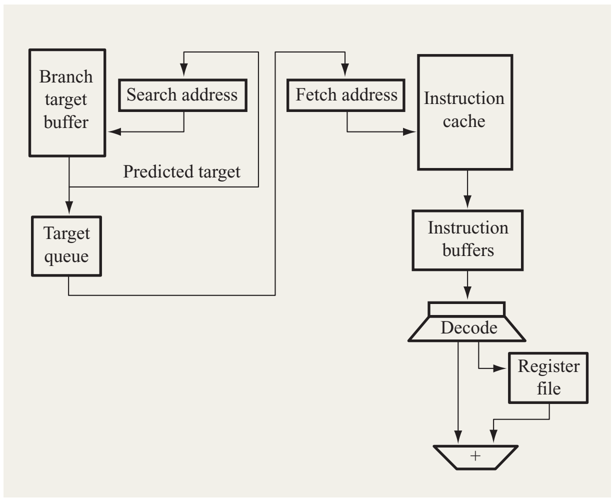

Figure 2

Branch prediction optimized for indirect branches.

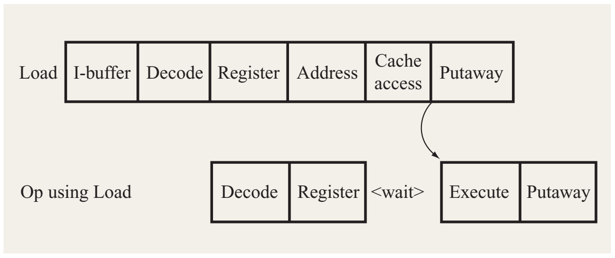

Figure 3

Instruction pipeline with address calculation in execution stage, showing cache access latency delay for dependent instruction.

decode time, and the target address being computed using information in the instruction fetch buffer.

In an S/390 processor, obtaining the branch target stream requires that the branch instruction be decoded (a more complex task than in most RISC ISAs), the base (and possibly index) general register accessed (resolving any interlocks with prior instructions), the effective address computed, and the cache accessed. The resulting latency for a taken branch, even if the direction is correctly predicted at decode time, is significant. This motivates a greater emphasis on predicting *both* the direction and the target address of each branch instruction, leading to a combined mechanism to predict the action completely for each branch rather than separate direction- and target-prediction mechanisms. This approach, illustrated in **Figure 2**, has been used in several S/390 processor designs [6, 9]. Since target addresses require more bits of storage than even the most complex forms of direction-prediction information, such a combined mechanism is somewhat larger than a direction-only prediction structure with the same number of branch entries. This size disadvantage is compensated somewhat by the opportunity to predict the target addresses of branches even while those branch instructions themselves are still being fetched from the cache. This allows the design to eliminate entirely the target-stream access latency for correctly predicted branches without requiring any special early decode or early address-generation logic.

Storage operand access

The handling of storage operands (as opposed to operands contained in registers or in the instructions themselves) is a distinguishing feature both among ISAs and among microprocessors. RISC ISAs typically limit storage access to a relatively small number of load and store instructions which transfer data between registers and storage, relying on separate register-to-register instructions to perform operations on the data in the relatively large (at least 32 general registers) register set. ESA/390, as is typical of CISC ISAs, includes a large set of register-to-storage, storage-to-storage, and immediate-to-storage instructions, which make use of operands taken directly from storage without first loading them into registers. The ESA/390 approach in many cases allows the same function to be performed with fewer instructions than with a RISC ISA, provided that these storage-operand instructions are used effectively. In order to exploit this to a performance advantage, an S/390 microprocessor must process these instructions efficiently. Among other things, this requires careful consideration of the storage-operand access design.

There are as many approaches to storage-operand access as there are microprocessor designs, but a key distinction among the various approaches is the point at which operand addresses are computed and operand accesses are initiated in the instruction pipeline. One method is to treat the operand address computation in the same manner as any other arithmetic operation, performing it in the normal execution stage, followed by the operand access itself. This results in a gap between the execution of a load instruction and the execution of an instruction using the value loaded; this “load-use latency” is a function of the time required to obtain data from storage. This approach is illustrated in **Figure 3**, which shows the core pipeline stages for a load instruction followed by an instruction which uses the value loaded; note that this two-instruction sequence, typical in RISC ISAs, is equivalent to a single register-storage instruction in a CISC ISA such as ESA/390.

An alternate design is to accelerate storage-address computations and storage-operand accesses in the pipeline so as to eliminate the load-use latency by making storage operands available to execution at the same point in the pipeline as register operands (assuming that the storage operands are found in the cache and encounter no delays). The elimination of the load-use latency is offset by the requirement that storage addresses be computed early in the pipeline, which introduces a new pipeline delay if the registers used to generate an address are being updated by a recent instruction. This address-generation interlock (AGI) applies whether the register is being updated by a load instruction or by an arithmetic or logical operation, and is a function of the distance in the pipeline between the address-generation and -execution stages.

Although there is some analogy between these two approaches and the CISC/RISC ISA debate, the two issues are in fact distinct: A number of CISC processors (including some for S/390) have been designed to perform address calculation at the same stage as execution, and RISC processors which follow this approach often include significant hardware dedicated to reducing the load-use penalty. Rather, the selection of one method or the other is a function of the frequency of different pipeline latency (interlock) conditions and the effective cost (in cycles) of each occurrence. That cost is a function of the software, the pipeline depth, the cache-access latency, and the ability of the pipeline to hide latency via buffering, prediction, out-of-order processing, special bypasses, or other techniques.

After weighing all of these factors, S/390 processor design tends to favor accelerating address calculation and operand access. One element of this preference is the need to execute storage-operand instructions well. Instruction pipelines, even those supporting out-of-sequence operation, run most efficiently when all common operations take the same amount of time, since variable latencies and durations complicate the scheduling of work for various resources. If storage-operand instructions are being executed frequently, requiring these to take significantly more cycles than register-operand instructions would disrupt the flow of instructions, affecting performance. For some compute-intensive benchmarks and workloads, the frequency of storage-operand instructions is low enough that optimizing for these instructions is not indicated, and may even be counterindicated if this optimization yields a deeper nominal pipeline in addition to the AGI penalties. More typical of the high-end server environment, however, are workloads in which relatively little computation is done on each operand fetched from storage. In this case, the load-use latency can dominate CPU performance, and the optimal design will be one which minimizes (or even eliminates) this penalty [10]. The pipeline shown in **Figure 4**, used in the current S/390 CMOS processors,

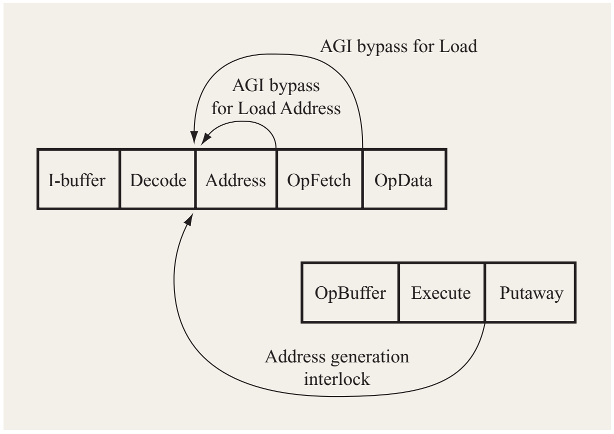

Figure 4

Instruction pipeline showing AGI interlock and AGI bypass paths.

reflects this optimization, with a load-use latency of zero cycles but an AGI penalty of up to four cycles. The AGI penalty can be mitigated via bypass logic and data paths for special cases, for example when the register value is being generated by a Load instruction (with no further computation) or by a Load Address instruction. This penalty can also be reduced by optimizing the code generated for S/390 to distinguish between address and other computations, and to schedule nondependent instructions in the AGI interval.

Cache design

The focus on the high-end commercial server market influences the cache design as well. S/390 systems typically operate at a very high degree of multiprogramming, with many tasks being managed at once by the OS/390 operating system. Furthermore, commercial transaction-processing and database programs tend to operate on a relatively large working set of data. Since the performance of any processor falls off rapidly when the operand data needed are not in the local cache, or when the virtual-to-real address translations needed are not in the translation lookaside buffer (TLB), S/390 processors place a high priority on minimizing the cache and TLB miss rates, even at the expense of more complex access logic. Not only does this imply a relatively large TLB and first-level (L1) cache, it also favors a set-associative rather than a direct-mapped design for both structures, greatly increasing the comparison logic required and coupling the output of this logic to the last stage of cache data selection. This complexity is justified, at least for S/390, by the tremendous reduction in TLB and cache miss rates, as illustrated in **Figure 5**.

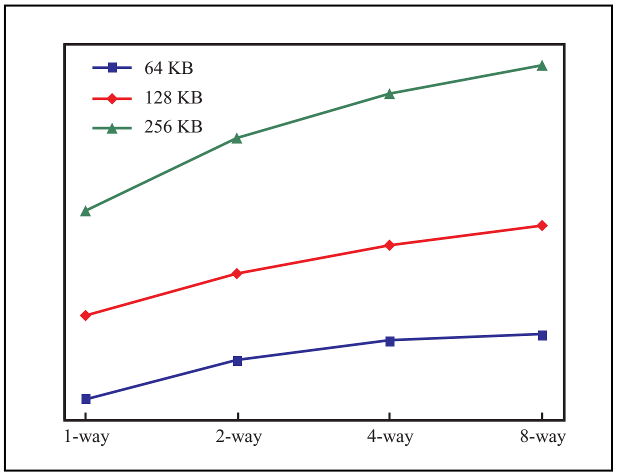

Figure 5

Effect of associativity on L1 cache performance.

One negative effect of such a cache design is that it yields a longer nominal access time for the hit case. This is further aggravated in S/390 by support for the arbitrary alignment of operands in storage, which adds more logic in the data path for these operands. These effects put upward pressure on the load-use penalty and help to tilt the balance in favor of the accelerated storage-operand access design described above. Similar effects in the instruction fetch path increase the latency there as well, adding to the value of predicting branch target addresses well ahead of branch-instruction decode, as noted earlier.

The cache design is also affected in fundamental ways by the combination of the ESA/390 storage architecture, the high-end server focus, and the RAS requirements. ESA/390 requires, among other things, that fetches and stores by all processors in a multiprocessor system appear to all processors to happen in the same order. In a large multiprocessor system, this argues for a write-back cache management scheme, in which cache blocks can be held in an L1 cache with "exclusive" rights, preventing other processors from accessing locations that are in the process of being updated without at the same time adding delay to every store operation. A simple write-back scheme, however, results in changed data being held in the L1 cache for an indefinite period of time. This creates a reliability problem, since if a processor fails while holding changed data, those updates are lost, possibly resulting in a system crash. Furthermore, cache arrays are especially prone to intermittent ("soft") single-bit errors; to prevent one of these errors from causing a system failure, error-correcting code of some form would be required in the L1 cache, increasing both the size and complexity of the L1 cache design. These RAS liabilities are avoided by using a write-through strategy, forwarding all stores immediately to the next level of cache, while managing the cache as if it were write-back [11].

Complex instruction execution

In addition to the moderately complex storage-operand instructions, the ESA/390 ISA includes a large number of instructions which perform significant program control or logical functions. Since such instructions cannot practically be implemented with hard-wired controls, some sort of internal code is required. In the past, high-performance S/390 designs have included a special-purpose execution engine (often VLIW-like in structure) for these functions. More recently, the drive to attain faster machine cycle times has led to a form of vertical internal code known as millicode [7]. This allows the complex ESA/390 functions to be executed with the same instruction pipeline as the simpler, hard-wired instructions. It carries with it, however, the requirement for additional registers for millicode use, and the need for quick transitions between ESA/390 and millicode modes of operation. Millicode also requires access to the full range of system control registers, tending toward a design in which these are mapped into a regular structure accessed by a small number of instructions rather than being scattered about the processor and accessed with *ad hoc* special functions. The use of millicode also adds another class of software for which the design must be optimized, since some commercial transaction processing workloads spend 10% or more of their time in millicode mode.

Pervasive functions

A different dimension of the "complex" part of the ESA/390 CISC ISA is found in the definitions of control functions which span the instruction set. Program event recording (PER), for example, requires that instruction, branch-target, and operand-store addresses be compared to a program-specified range, which favors the use of separate dataflow elements dedicated to address handling. More generally, ESA/390 requires that all program exceptions be indicated by precise interruptions, with strictly defined bounds on what the processor may do in the presence of exceptional conditions. This mandates a comprehensive mechanism for suppressing updates to the processor state in those circumstances, which argues for at least one control point in the pipeline which is downstream from all exception checking and upstream from all persistent state updates. Here again, the acceleration of storage accesses in the pipeline proves advantageous, as this facilitates checking of all operand-access exceptions prior to the start of execution for a given instruction, even in the case of instructions with long storage operands, without adding to the overall instruction latency. An ESA/390 processor is also required to handle precisely any operand stores into the instruction stream, which requires that store-address and instruction-fetch addresses be compared and store/I-fetch conflicts resolved, even when the store is into the location of the instruction immediately following. This affects the structure of the instruction buffer and of the pipeline instruction queue, and the execution of store instructions. In none of these cases is ESA/390 unique, and in each one design solutions are known for a wide variety of microarchitectures; taken in combination, however, these often prove to be the deciding factors in choosing among alternate designs for specific functions and hardware units.

One requirement which is unique to S/390 is support for interpretive execution. This facility is the basis for both the VM/ESA* operating system and the PR/SM logical partitioning facility. PR/SM allows multiple workloads to be consolidated efficiently on a single S/390 system, greatly enhancing the customer value of these systems; VM/ESA is a key part of many S/390 customers' overall operations, allowing great flexibility in the number and type of operating system images. Performance in interpretive execution mode, including the performance of VM/ESA running in a PR/SM logical partition, is therefore crucial to the position of S/390 in the high-end server market. This dictates that support for interpretive execution be designed into the processor from the start, including multiple copies of the control state (control registers, timing facilities, etc.) for different levels of emulation, support for multiple levels of address translation, and integration of emulation-specific special-condition checking into the normal instruction controls. While this adds to the overall complexity of the control structures in an S/390 processor, and thus implicitly affects the cost of implementing some microarchitectural features, it enables a level of efficiency at the system level for virtual machine and logically partitioned operation which is unmatched by any other platform.

An example of the utility of this feature of S/390 can be seen in the implementation of support for the open-source Linux operating system on S/390. By providing a means to present an architecturally complete "virtual machine" image, the combination of PR/SM and VM/ESA allows for multiple (even many) copies of (for example) Linux and OS/390 to share the same S/390 system, each with the appearance of executing on its own machine, architecturally protected from one another, and yet able flexibly to exploit the full power of the underlying hardware.

Error detection and recovery

The use of S/390 systems for enterprise-scale mission-critical applications places a premium on all aspects of system reliability, availability, and serviceability. This drives a wide range of design decisions at all levels of system design, as has been described in [5]. In the realm of microprocessor design, the most crucial aspect of this is that *all* hardware errors must be detected before they have led to any unrecoverable loss of data. The requirement of recoverability—that is, the capability to continue processing after a fault has been detected with no customer impact—greatly narrows the range of design options for error detection. The requirement for high-performance (and therefore high-frequency) operation further constrains this space, because maximum performance implies minimal margins in cycle time and circuit parameters, leaving little room for the overhead of checking circuits. These considerations have led to an S/390 microprocessor design [7–9] in which the units requiring complex dataflow are duplicated for error detection, with all operations being performed twice in parallel and all resulting updates to the processor state being checked. This provides complete fault coverage in these units without incurring any overhead for parity prediction or similar schemes. The area overhead from such a design is kept at much less than 100% by using more conventional parity checking in those units (such as caches) in which the dataflow is predominantly byte-coherent, so that minimal parity-generation logic is needed.

Recoverability implies that errors are detected at a point in time when operation can still be resumed from a state that is known to be error-free. For comprehensive RAS, this must generally be done at multiple levels within the overall system, including hardware, LIC, system software, and application programs, in each case detecting and recovering from errors in that level. In some cases errors occurring at one level must be passed to another level for resolution (e.g., using LIC or system software to recover from hardware failures); doing so, however, inevitably increases the complexity and reduces the effectiveness of recovery, with a corresponding negative impact on system RAS. For S/390 microprocessor designs, therefore, the goal is to detect and handle hardware faults at the lowest practical level (i.e., hardware, then LIC, then system software). To this end, the S/390 CMOS microprocessors since G4 have maintained a complete set of processor-state data in a structure that is protected by error-correcting code (ECC). Updates to this state structure are made only when the updates have been checked for correctness (by comparing the versions generated by the two copies of execution hardware). Data paths are provided by which the state can be propagated systematically to all parts of the microprocessor via a hardware-controlled refresh operation. When a fault is detected in the hardware, all operations which have not been completed and validated are discarded along with all buffer contents (including the cache), the processor state is refreshed from the known good state, and operation is resumed at that point. If the hardware fault does not persist, normal operation continues as if no fault had occurred. Since the protected and refreshed state includes that of all internal and LIC architectural facilities, this sequence allows for full recovery even within millicode routines. In the event of a persistent fault, in which it is not possible to continue normal operation, this design allows for the state from the failing processor to be extracted via a service processor and injected into a spare processor in the same system; the spare processor assumes the role of the failed processor and continues from the extracted state, transparently to software and (in most cases) to millicode as well.

The need for serviceability influences the microprocessor design as well, in that this motivates a design in which LIC (specifically, millicode) has broad capabilities for taking on additional functions. The S/390 CMOS designs provide millicode with both read and update access to the processor state structure used for recovery, allowing millicode to perform almost any function imaginable within the scope of the microprocessor. Besides providing a convenient way to implement newly defined functions on an existing design, this enables a powerful class of "work-arounds" for any design errors discovered after the hardware has been manufactured, and even after the product has been shipped to customers. When combined with hardware controls which can force specific operations or sets of operations to be performed by millicode rather than hardware (which controls are themselves included in the millicode-accessible processor state), this design enables effective fixes to be deployed without having to change any hardware.

Conclusion

We have described the manner in which a number of specific requirements have influenced the design of S/390 microprocessors in IBM, particularly as reflected in the S/390 CMOS processors starting with G4. While not one of these requirements is strictly unique to S/390, and while many designs are possible which satisfy these requirements, their importance in the S/390 space has led to a microprocessor design point which in many particulars differs from that chosen for other products and platforms. As the art of microprocessor design progresses, and as market and technical requirements for both S/390 and other platforms change, the differences apparent now between S/390 and others can be expected to change as well. The challenge for each design team, regardless of platform, will be to meet all of these changing requirements in each new design while maximizing performance and minimizing both cost and design time.

*Trademark or registered trademark of International Business Machines Corporation.

References

1. IBM, *Enterprise Systems Architecture/390 Principles of Operation*, Order No. SA22-7201-03, September 1996; available through IBM branch offices.
2. IBM, *z/Architecture Principles of Operation*, in press.
3. P. Mak, M. A. Blake, C. C. Jones, G. E. Strait, and P. R. Turgeon, "Shared-Cache Clusters in a System with a Fully Shared Memory," *IBM J. Res. Develop.* **41,** No. 4/5, 429–448 (1997).
4. P. R. Turgeon, P. Mak, M. A. Blake, M. F. Fee, C. B. Ford III, P. J. Meaney, R. Siegler, and W. W. Shen, "The S/390 G5/G6 Binodal Cache," *IBM J. Res. Develop.* **43,** No. 5/6, 661–670 (1999).
5. L. Spainhower and T. Gregg, "IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective," *IBM J. Res. Develop.* **43,** No. 5/6, 863–874 (1999).
6. J. S. Liptay, "Design of the IBM Enterprise System/9000 High-End Processor," *IBM J. Res. Develop.* **36,** No. 4, 713–731 (1992).
7. C. F. Webb and J. S. Liptay, "A High-Frequency Custom CMOS S/390 Microprocessor," *IBM J. Res. Develop.* **41,** No. 4/5, 463–473 (1997).
8. T. J. Slegel, R. M. Averill III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. McDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb, "IBM's S/390 G5 Microprocessor Design," *IEEE Micro* **19,** No. 2, 12–23 (1999).
9. M. A. Check and T. J. Slegel, "Custom S/390 G5 and G6 Microprocessors," *IBM J. Res. Develop.* **43,** No. 5/6, 671–680 (1999).
10. T. Horel and G. Lauterbach, "UltraSPARC-III: Designing Third-Generation 64-Bit Performance," *IEEE Micro* **19,** No. 3, 73–85 (1999).
11. K. M. Jackson and K. N. Langston, "IBM S/390 Storage Hierarchy—G5 and G6 Performance Considerations," *IBM J. Res. Develop.* **43,** No. 5/6, 847–854 (1999).

Received February 15, 2000; accepted for publication November 8, 2000

Charles F. Webb *IBM Server Group, 2455 South Road, Poughkeepsie, New York 12601 (cfw@us.ibm.com).* Mr. Webb is a Distinguished Engineer in z-Series processor design in Server Group development. He received his B.S. degree in 1982 and his M.Eng. degree in 1983, both from Rensselaer Polytechnic Institute. He joined IBM in 1983 at the Product Development Laboratory in Poughkeepsie, where he has remained since. Mr. Webb has worked on the ES/9000 processor, the S/390 G4 and G5 CMOS processors, and the *eServer* z900 processor in the areas of performance analysis, architecture, and design. He has received thirteen IBM Invention Achievement Awards, three IBM Outstanding Innovation Awards, and an IBM Corporate Award. He is a member of the IBM Academy of Technology and the IEEE Computer Society.