SCISM: A scalable compound instruction set machine

by S. Vassiliadis
B. Blaner
R. J. Eickemeyer

In this paper we describe a machine organization suitable for RISC and CISC architectures. The proposed organization reduces hardware complexity in parallel instruction fetch and issue logic by minimizing possible increases in cycle time caused by parallel instruction issue decisions in the instruction buffer. Furthermore, it improves instruction-level parallelism by means of special features. The improvements are achieved by analyzing instruction sequences and deciding which instructions will issue and execute in parallel prior to actual instruction fetch and issue, by incorporating preprocessed information for parallel issue and execution of instructions in the cache, by categorizing instructions for parallel issue and execution on the basis of hardware utilization rather than opcode description, by attempting to avoid memory interlocks through the preprocessing mechanism, and by eliminating execution interlocks with specialized hardware.

Introduction

Improvements in the performance of computer systems relate to circuit-level or technology improvements and to organizational techniques such as pipelining, cache memories, out-of-order execution, multiple functional units, and exploitation of instruction-level parallelism. One increasingly popular approach for exploiting instruction-level parallelism, i.e., allowing multiple instructions to be issued and executed in one machine cycle, is the so-called superscalar machine organization [1]. A number of such machines with varying degrees of parallelism have recently been described [2, 3]. The increasing popularity of superscalar machine organizations may be attributed to the increased instruction execution rate such systems may offer, concomitant with technology improvements that have made their organizations more feasible.

Neither superscalar machines nor scalar machines, i.e., machines that issue and/or execute a single instruction per cycle, may necessarily exploit all of the potential performance improvements that their organizations promise. In superscalar machines, the level of parallelism achieved may be less than anticipated for a variety of

reasons, including data dependencies (interlocks), branch instructions, and precise interrupt handling. Data dependencies are particularly detrimental to superscalar machines because they force serial instruction execution—the superscalar machine reverts to scalar execution mode, in which additional pipeline dead cycles (bubbles) that are intrinsic in the underlying scalar implementation may be incurred. Thus, much of the performance gain promised by a superscalar organization may be lost [1, 4]. Furthermore, implementing and, in particular, controlling multiple functional units is not without cost—a cost that may not be worth paying if little is to be gained.

To quantify this cost, consider a machine organization designed to issue n instructions. Instructions are fetched into an instruction stack (i-stack) or window, and at instruction decode time (i.e., from the instruction stack or window) a decision is reached as to whether or not a given n-tuple or, simply, *group* of instructions may be executed in parallel. This decision-making process, referred to as *preprocessing*, is based on the opcodes of the instructions, which specify the hardware each of the n instructions will utilize, and on the dependencies existing among the instructions. (Typically, to avoid creating bubbles internal to the group of instructions, no dependencies may exist if the instructions are to be executed concurrently.) Then the instructions are issued and executed. This organization has the following characteristic: The amount of time required to allocate n instructions to m appropriate functional units (sometimes called "instruction split" [5]) during instruction decoding increases as m increases. Yet at the same time it may seem very desirable to increase m in order to increase concurrency. For a complex instruction set architecture, e.g., ESA/370™ [6], decoding and analyzing the dependencies among two or more instructions takes substantially more hardware and time than it would for a simpler architecture,[1] since the task is considerably more difficult.

In some machines, preprocessing is based on the instruction opcode description [2]. This implies that in order to exploit the existing parallelism in an instruction stream, it is necessary to implement a prohibitively large number of rules to control parallel instruction issue. To clarify this point, consider a superscalar machine designed to execute at most two instructions in parallel. Furthermore, assume that there are W instructions in the machine instruction set. To maximally exploit instruction-level parallelism, $W \times W$ rules must be implemented in the instruction decode hardware to determine whether or not a given pair of arbitrary instructions may be issued in parallel. It is our contention that for a nontrivial instruction set, implementing this number of rules and using them to gate the issue/not-issue decision is prohibitive with respect to hardware, circuit complexity, and delay. Indeed, present-day superscalar machines using this approach consider only a few combinations of instructions for parallel execution. For example, the superscalar machine presented in [2] considers only 38×53 rules, and it allows 38 and 53 specific instructions out of the instruction set to be the first and the second instructions, respectively, in an instruction pair issued in parallel. Furthermore, some other superscalar machines, in order to restrict the number of rules required for parallel execution, may issue instructions in parallel only in very specific circumstances [3] and therefore do not exploit the instruction-level parallelism present in most programs to the highest degree possible. Additionally, in a group of n instructions, where n is sufficiently large (perhaps 3 or more), it is very likely that dependencies will exist between instructions. These dependencies will prevent concurrent execution, forcing a serial execution instead, and will therefore limit the performance gain of the superscalar machine.

In superscalar machines that are currently available, preprocessing is a first-in-first-out (FIFO) operation, out of the instruction buffer (also referred to as the instruction stack or window), which attempts to decide "on the fly," i.e., at instruction issue/decode time, whether two or more (up to n) instructions may be issued and executed in parallel. Consequently, the *scope* of preprocessing is restricted to n. This too restricts parallelism, since certain dependencies could be avoided if alternate groupings of instructions could be considered. For example, a load–use [8] may require a cycle of delay between the load from memory and subsequent use. If we assume $n = 2$ and FIFO preprocessing, the load and use are serialized because of the required delay. Suppose, however, that the scope of preprocessing is greater than n. Then the possibility exists for a more optimal grouping, i.e., pairing the load with the previous instruction and the use with the instruction after it, thereby maintaining the required load–use delay but replacing a bubble cycle with useful work. While a larger scope of examination could be postulated for a superscalar machine, there are two consequences of increasing the scope that penalize potential performance gains. The first is that to analyze more instructions requires more time, i.e., circuit delay, and therefore the machine cycle time may be increased prohibitively, thus penalizing the execution of all instructions. A second consequence, related to the first, is that the time required may be so great that it may be necessary to add an extra stage to the pipeline, which penalizes the execution rate each time the pipeline is drained, when, for example, the outcome of a branch instruction is mispredicted. Thus, the effects of broadening

[1] In this paper, the term *architecture* denotes the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior of the machine, and is distinct from the organization of the dataflow and physical implementation of the machine [7].

the scope of examination are diminished in the presence of branch instructions.

The branch problem is well known, and a variety of techniques have been developed to alleviate the detrimental effects of branches on performance. These techniques include various branch prediction schemes [9, 10] and methods for early branch outcome determination [11, 12]. In superscalar machines the problem is magnified, since the frequency at which branch instructions enter the pipeline is increased. To illustrate, assume a superscalar machine designed to issue and execute three instructions per cycle, and assume that branch instructions comprise 25% of the instruction stream. Clearly, it is of interest to issue three instructions every cycle to maintain maximum performance. If it were possible to sustain this rate of execution, which is not unreasonable, then at this branch frequency a branch would have to be processed every other cycle. Consequently, an efficient branch-handling mechanism would be needed to minimize pipeline disruptions. Even with such a mechanism, once a group of instructions has been executed, all information from preprocessing that group is lost. Branches influence this loss in two instances. First, consider the case in which the outcome of a branch is mispredicted. Even if the correct sequence has been executed previously, no information exists from previous preprocessing. This implies that either the incoming correct instruction stream is analyzed on the fly, thus potentially lengthening the duration of the branch-induced disruption, or the execution is serialized (instructions starting from the first instruction of the correct stream are issued one at a time until the rate of preprocessor output is at least equal to the rate at which instructions are being consumed). Depending on implementation, this point may never be reached if the incidence of taken branches is high. Second, consider the case in which a branch instruction is correctly predicted. Furthermore, to possibly avoid a cycle time increase and to maximize parallelism, assume that some scheme has been employed to avoid strict FIFO preprocessing, perhaps by preprocessing instructions early in the i-stack. If no branches are taken, the instruction text present in the i-stack can be successfully preprocessed in advance. However, branches are frequently taken, and, irrespective of prior executions, the branch target instructions must be preprocessed again, limiting the effectiveness of early preprocessing. The conclusion to be drawn from these two cases is that associating preprocessing with instruction fetching may not result in the best possible performance, and the former must somehow be detached from the latter.

In summary, currently available superscalar machines may exhibit one or more of the following deficiencies that limit their execution rates:

1. Increased cycle time caused by the additional circuit delay required to allocate instructions to functional units "on the fly."
2. Additional pipeline stages required to accommodate instruction preprocessing.
3. Limited numbers of instructions that can be issued and executed in parallel (in order to avoid increasing either the cycle time or the number of pipeline stages).
4. Prohibitive circuit counts, hardware complexity, cycle time growth, and/or a longer pipeline when more comprehensive groupings of instructions are considered.
5. Pipeline bubbles induced by data dependencies.
6. Failure to overcome data dependencies by broadening the preprocessing scope.
7. Continuous, nonpermanent preprocessing resulting in performance degradation.
8. Potential loss of parallelism due to the higher incidence of branch instructions entering the pipeline.

We present a new machine organization called a *scalable compound instruction set machine* (SCISM), also referred to as a compound instruction set machine[2] and as a superscalar compound instruction set machine, which incorporates mechanisms that either solve or alleviate the problems described previously. The proposed organization results in improvements in performance and hardware requirements when compared to existing superscalar machines, and does so without precluding other improvements that may result from out-of-order execution, static scheduling, pipelining, etc. Additionally, the machine organization may use compiler technology, e.g., instruction scheduling [18], to further improve performance. Lastly, the organization results in designs that are fully compatible with any given architecture.

This paper is organized as follows. The Concepts section presents the fundamental attributes of the SCISM organization. The Organization section describes the machine organization that employs these concepts. Finally, the Evaluation section presents simulated performance results for a SCISM processor.

Concepts

The discussion in this section and the sections to follow presents some of the general concepts proposed in the report cited by footnote 2 and describes in depth the development of those concepts that resulted in an experimental design in the IBM Glendale laboratory in Endicott, New York. This section concentrates on the categorization of instructions, dependency resolution, and preprocessing—three fundamental mechanisms of the SCISM organization.

[2] S. Vassiliadis, "Compound Instruction Set Machines," private communication, 1989.

To avoid the rule limitation imposed by the approach used in [2], the grouping of instructions that can be issued and executed in parallel must be based on hardware utilization rather than opcode description. This hardware utilization basis is formed by the following characteristics:

1. Instructions are partitioned into categories.
2. All instructions in a category are viewed as "unique" instructions.
3. Differences among members in a category are "trivial" and are resolved by the hardware.
4. An instruction may be in a particular category if and only if it uses the same hardware units as all others in that category.

All instructions that do not meet these definitions can either be assigned to some number of individual categories or be lumped together in a single category. The latter simplifies the implementation of preprocessing by minimizing the number of instruction categories.

An obvious implication of these definitions when applied to a given instruction set is that the number of rules required to group the instructions for parallel execution depends on the number of categories into which the instruction set has been partitioned, rather than on the number of individual instructions in the instruction set. The intuitive reasoning behind such partitioning is that there are a limited number of functional units present in an implementation and that such units operate on a multiplicity of instructions. For example, an arithmetic logic unit (ALU) executes all add, subtract, add logical, compare, compare logical, logical AND, and logical OR instructions, and so forth. Certainly these instructions differ from one another, but nevertheless such differences are trivial. ("Trivial" in the context of this discussion indicates that while there is a distinct operation associated with each instruction, such a difference is resolved by some simple control signal or by some minor modification of the hardware to accommodate the operation. For example, in two's-complement arithmetic, an addition differs from a subtraction in that the latter requires inversion of the subtrahend and the addition of 1, typically provided by injecting a "hot 1" carry-in to the ALU, together with the control information indicating that a subtraction is to be performed rather than the addition.) Additional functional units that may be included in a hardware implementation include the floating-point adder, floating-point multiplier, branch unit, address generation unit, shifter, etc. This implies that the definitions postulated for instruction categorization will be successful in reducing the complexity of instruction preprocessing, since many instructions may be considered as a "single" instruction.

As indicated in the Introduction, performance gains in superscalar machines can be lost to data dependencies (interlocks) between instructions. We note here that such dependencies can be divided into two categories, namely execution interlocks and memory interlocks. Execution interlocks occur within a functional unit, e.g., register write–read interlocks, and memory interlocks occur between a functional unit and memory, e.g., the load–use interlock. To eliminate execution interlocks, an implementation must incorporate both multiple execution units and multi-operand execution units. The requirement for multi-operand execution units necessitates the design of units that produce architecturally correct results but do not extend the cycle time or require prohibitive quantities of hardware.

To illustrate the execution interlock elimination mechanism, which we call *interlock collapsing* hardware, we assume, with obvious generalizations, that the instruction set is that of the IBM System/370™ architecture. Furthermore, let the number of instructions that can be executed by the interlock collapsing hardware unit be limited to two. There are two reasons for this choice. First, workload analysis has shown that the likelihood of encountering in a program three or more adjacent interlocking instructions that require the same hardware unit is very small. Second, it has been proven that the implementation of a two-instruction interlock collapsing hardware unit does not extend machine cycle time [13]. While a cursory analysis suggests that general forms of interlock collapsing hardware units, e.g., a unit that can execute interlocked shift and add instructions, can cause prohibitive circuit delays, the most frequent interlocking instructions have simple operations that can be collapsed, e.g., arithmetic operations, logical operations, register transfer operations, address generation, and branch outcome determinations. These simple operations are the focus of the SCISM interlock collapsing hardware.

When considering the hardware necessary to perform these simple operations, one might correctly conclude that an interlock collapsing ALU would be more complex by virtue of the sheer amount of function it must provide: It operates with both two's-complement numbers and unsigned numbers, and performs arithmetic and logical operations and register transfer operations with some of the instructions, some (but not all) of which set condition codes, cause overflows, and so on. Furthermore, although many instructions utilize an ALU and may therefore be lumped into one category, each instruction within the category utilizes the ALU in a slightly different manner. For example, in the IBM System/370 instruction set, register-to-register fixed-point instructions that require an ALU and have the same execution sequence can be subdivided into the following groups:

- Arithmetic, e.g., AR, LPR.
- Arithmetic logical, e.g., ALR, SLR.
- Logical, e.g., XR, NR.
- Logical compare, e.g., CLR, CR.
- Load register, e.g., LR, LTR.

Instruction categorization as it has been defined previously suggests that these five subcategories could be considered as a unique SCISM category. This, however, is an implementation choice: It may not be necessary for all of these subcategories to be lumped into a single SCISM category (depending on the frequency of individual instructions in a program and the hardware constraints).

The assumption of pairwise instruction execution dictates the existence of two ALUs that provide the capability for concurrent execution of two instructions. Given that two sequential instructions can have an interlock only from the first instruction to the second, not from the second to the first, the ALU that executes the first instruction of the pair is a traditional 2-to-1 ALU. Since the second instruction may be dependent on the execution of the first instruction, in order to be able to execute both instructions in parallel the second ALU must be capable of performing 3-to-1 ALU operations. To illustrate, consider the following instruction sequence:

```
SR R1,R2
AR R1,R3
```

Assuming that R*j* denotes the content of register *j*, to execute this instruction pair in parallel the first ALU must perform the SR (subtract register) operation R1 = R1 − R2 and the second ALU must perform the AR (add register) operation R1 = R1 + R3 = (R1 − R2) + R3, i.e., a 3-to-1 operation.[3]

Any design of a 3-to-1 ALU must be proven architecturally correct and must be assessed as to its feasibility as constrained by hardware and cycle time considerations. For a set of 3-to-1 ALU operations, which is defined by both the instructions included in the ALU operations category and the permissible pairwise operations the ALU may perform, the following two statements hold true (and are proven elsewhere) [13–15]:

- A 3-to-1 ALU can be designed that guarantees architectural compliance by producing the correct result, correctly setting condition codes, and detecting overflow for the locus of operations it performs [13, 14, 16].
- The design of the 3-to-1 ALU requires no more delay than a 3-to-1 binary addition, and no more stages than the design of a 2-to-1 ALU usually implemented to perform fixed-point operations in RISC and CISC architectures [13, 15, 16]. The implication here is that its implementation does not increase the machine cycle time [13, 15, 16].

The feasibility of the 3-to-1 ALU suggests that other, adder-related, interlock collapsing hardware is possible,[4] e.g., address generation interlock collapsing units.

Regarding memory interlocks, it can be stated that in general they cannot be feasibly resolved. While some memory interlocks can be alleviated with the use of techniques such as pipeline forwarding and out-of-order execution, the problem still remains. In addition to these techniques, the SCISM approach allows memory interlocks to be avoided by using one of two additional mechanisms: The first mechanism requires the design of a specialized unit and the preprocessing of load instructions. (It is described in [17] and is not discussed further here.) The second mechanism involves grouping a memory-referencing instruction with previous instructions when possible. To accomplish this, preprocessing is removed from the i-stack or window to one of several locations (to be discussed later), allowing instructions to be analyzed with a broad scope. As is shown in the next section, this organization yields permanent, optimized preprocessing, obviating the need for preprocessing at instruction fetch/issue/decode time. By so doing, it is not only possible to broaden the scope of preprocessing, but it also becomes feasible to optimize preprocessing to achieve the highest degree of instruction-level parallelism that can exist in a given program, to avoid instruction-split cycle time or pipeline penalties by preallocating instructions to functional units, and to avoid the other limitations associated with preprocessing at instruction fetch/issue/decode time.

Organization

At its highest level of abstraction, the operation of the SCISM machine organization incorporating the concepts described previously can be represented by **Figure 1**. In the figure, a "program" is provided as an input to a

[3] Note the existence of a degenerate case requiring a 4-to-1 ALU. Consider the instruction sequence

```
SR R1,R2
AR R1,R1
```

For the pair of instructions to be executed in parallel and with only one pass through the ALU, the second ALU must perform the operation R1 = (R1 − R2) + (R1 − R2). Given the added hardware complexity and circuit delay required to build such an ALU and the low frequency of such interlocks, little merit is found for including such an ALU in an implementation. Note also that such operations may not even require an interlock collapsing ALU for execution: In the example, bit shifting and subtraction suffice.

[4] Clearly, other interlocks can be collapsed. For example, consider the following sequence of System/370 floating-point instructions:

```
LDR R1,R2
MD  R1,D2(X2,B2)
```

In the first operation, LDR, the second operand R2 is placed unchanged at the first operand location R1. The second operation, MD, multiplies the operand contained in register R1 with an operand contained in a memory location. Obviously, the LDR is a register transfer instruction and the MD in effect operates on the operand contained in register R2. Such an interlock can easily be recognized and eliminated. The collapsing of this interlock, and others, is trivial when compared to the interlock collapsing ALU and is not considered further.

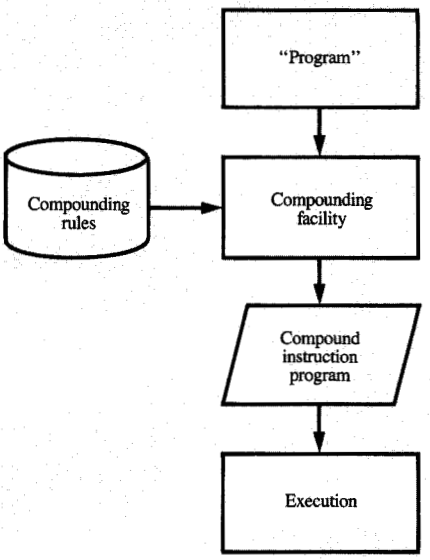
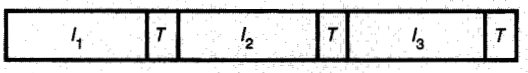


Figure 1

Abstract SCISM representation.



I_j is in original form

T = tag

Figure 2

Maximum compound instruction format for the compounding of at most three instructions.

compounding facility (footnote 2). The compounding facility or preprocessor examines the instruction stream with an implementation-dependent predetermined scope and produces a *compound instruction program.*

The compound instruction program is based on a set of rules which reflect the system architecture, the hardware organization, and the permissible parallel execution between categories of instructions. These rules are hereafter referred to as *compounding rules*. The program produced by the compounding facility can then be executed directly by a compound instruction execution engine, which considers a compound instruction as a single instruction. A compound instruction reflects the parallel issue of instructions; it comprises some number of independent instructions or interlocked instructions, the latter provided that the interlocks are of a form that can be collapsed by the execution hardware.

As indicated in the previous paragraph, the compound instruction contains information pertinent to the parallel issuing and execution of instructions. In general the information, incorporated in the compound instruction in the form of decoding or tagging, indicates that compound instructions are "free of hazards," and that functional units required for the execution of an instruction are available when necessary. Instructions composing a compound instruction need not be consecutive, allowing for out-of-order issue. Additional information, related for example to branch prediction, functional unit allocation and control, or routing information can also be incorporated in the decoding or tagging of compound instructions. We note here that the choice between decoding instructions and tagging them depends entirely on the architecture and the implementation constraints. For reasons that become obvious later in this section, tagging of instructions is mandatory for architectures that allow variable-length instructions, or that allow data to be intermingled with instructions. In the remainder of this paper, we assume that the compounding information is in the form of tags rather than decoding. Additional discussion of the tags can be found in [18]. The tag identifies the boundaries between single and compound instructions. For example, **Figure 2** shows the format of a maximal compound instruction, if it is assumed that up to three instructions may be included in a compound instruction.

In the figure, the instruction I_j is in its original form (implying full compatibility with the base architecture), and T is the tag. As indicated earlier, the tag may contain as little or as much additional information as deemed efficacious for a particular implementation. In the rest of the presentation, for simplicity of exposition, we discuss only the information necessary to determine a compound instruction. To determine the compounding of three instructions, two bits are required, denoted as t_0 and t_1. These bits are required for delimiting compound instructions, with 00 representing single instructions and 01 and 10 representing two- and three-instruction compound instructions, respectively.

The two control tag bits t_0 and t_1 merit further discussion. One of the fundamental properties of the SCISM machine organization is that it enables preprocessing to be detached from instruction issue/decode. To do this, the compounding of instructions must be "permanent," with permanence being dictated by the location of the compounding facility. For example, the compounding facility may be a software facility—perhaps a

post compiler [19]—or it may be a hardware facility in the form of a hardware preprocessor located, for example, between the cache and the memory subsystems. In this hardware preprocessor, the "program" to be compounded is the stream of instruction text that is fetched during the servicing of a cache miss and preprocessed to produce an optimized stream of compound instructions, i.e., a stream of instructions with their tags. The compounding information in the tags remains intact as long as the line resides in the cache, and is thus relatively "permanent." If the line should be removed from the cache for any of a number of reasons, the associated tags become invalid and the line must be preprocessed again should it be required at some later time.

As long as valid tags are available, the SCISM engine

1. Fetches and executes compound instructions.
2. Executes maximally n instructions in parallel, where n is the degree of instruction-level parallelism the hardware can support.
3. Maintains "permanent" compounding during sequential and nonsequential (branches, interrupts) execution.
4. Produces correct results in the presence of certain execution interlocks, e.g., proper two's-complement additions with detection of overflows as defined by two's-complement arithmetic.
5. Keeps intact the architectural behavior of the machine.

The permanence of compounding in SCISM during execution is maintained with the help of the tags, since an instruction is either executed sequentially or is a branch target, and in either case the compounding remains intact. The case in which an instruction is a branch target requires further explanation, because typically there can be no guarantee that a branch into the middle of a compound instruction will not occur. This is readily handled by hardware if, on a branch target instruction fetch, it

1. Fetches a number of bytes equal to the maximum-length compound instruction.
2. Identifies the end of the branch target compound instruction by locating the compound instruction delimiter for the next compound instruction.

Figure 3 illustrates this situation. In this figure, the maximum length of compound instructions (CI) is three, and the T field associated with each instruction has been reduced to the t_0 and t_1 bits only. Instructions within the mth compound instruction, CI^m, are denoted by I_r^m, where $r = 1$ for the first instruction in CI^m, $r = 2$ for the second instruction in CI^m, and $r = 3$ for the third instruction in CI^m. The second instruction of CI^1 is a branch instruction, B_2^1, that, for simplicity, is considered to have two possible target paths, a and b. The a path branches to the middle of

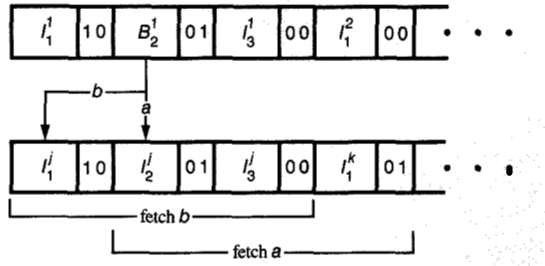


Figure 3

Branching to the "middle" of a compound instruction.

CI^j, while the b path branches to the beginning of CI^j. If the branch is to follow path a, the hardware fetches the maximum-length compound instruction, i.e., three instructions, then executes I_2^j and I_3^j as a compound instruction. The remainder of the fetch, I_1^k, is recognized to be the beginning of a new compound instruction and is held in reserve while the rest of CI^k is fetched for subsequent execution. If the branch instruction takes the b path to the beginning of CI^j, the hardware again fetches the maximum-length compound instruction, yielding, in this case, a complete compound instruction, I_1^j, I_2^j, and I_3^j. Execution of that compound instruction proceeds directly. The width of the tag required for compounding and used in this discussion is equal to $\log_2 n$, n being the maximum number of instructions that can potentially be compounded. Other tagging mechanisms are also possible [19].

The compound facility or preprocessor can be located in the software [19], in main memory [20], or in a cache. For this discussion and for our System/370 SCISM design, we have assumed that the preprocessor resides in the cache. The cache preprocessor has the following interesting properties:

1. Preprocessing occurs only at the cache miss rate and is thus infrequent.
2. Overhead for tag storage is added only to the cache memory and to the instruction fetch and issue hardware.
3. Architectural idiosyncrasies (discussed later in detail) such as writing into the instruction stream, data intermingled with instructions, and variable-length instructions can be handled inexpensively in terms of hardware and performance.

A computing system comprising an instruction compounding unit (ICU), a compound instruction cache

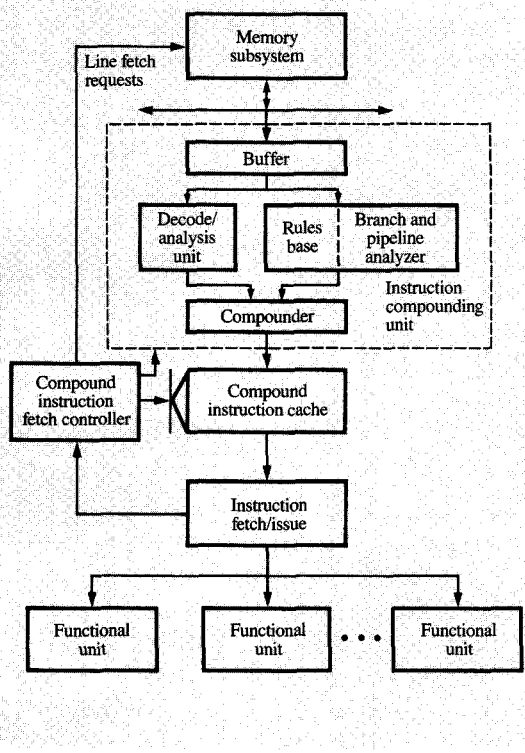


Figure 4

SCISM with cache preprocessor.

(CIC), and a number of functional units is shown in **Figure 4**. The preprocessing function is performed by the ICU in combination with the compound instruction fetching controller (CIFC) sequential machine, which oversees the entire process of supplying compounded instructions to the functional units, requesting line fetches from the memory subsystem, and other tasks which are discussed later. The ICU is further divided into a buffer, decode/analysis unit, rules base, branch and pipeline analyzer, and compounder. The buffer acts as a staging area between the memory subsystem and the CIC. The ICU performs the preprocessing function on instructions in the buffer, ultimately producing a stream of instructions with their tags. Clearly, the wider the buffer is in units of instruction width, the wider the scope of preprocessing, and the more opportunity there is for optimizing compounding across a given number of instructions. For example, suppose the buffer could contain two instructions, and at some point contained instructions i_1 and i_2 in one machine cycle and then i_3 and i_4 in the next. Also assume that two instructions may be compounded for parallel execution. Upon analyzing i_1 and i_2, the ICU determines that they cannot be compounded and must therefore be written to the CIC as noncompounded instructions. In the next cycle, the ICU analyzes i_3 and i_4 and makes the same determination. Now suppose that the buffer can instead contain four instructions, and can thus contain i_1, i_2, i_3, and i_4 simultaneously. The ICU can now also consider i_2 and i_3 for compounding, and if this pair does compound, the result is a performance improvement over the former case. The choices of buffer size and the form and extent of scope are engineering decisions that must trade off the advantages offered by a buffer of a particular size and a scope having a particular form and extent against the cost of implementation. Intuitively, the buffer size is a function of the number of instructions that constitute a single compound instruction and of the form and extent of the scope of examination. The extent of scope may range from a few instructions, to a whole cache line, to adjacent cache lines. The form of scope may range from serial examination, to a sliding window, to multiple passes, and so on. It may be that increasing the buffer size beyond a certain value or making the scope overly complex may produce diminishing returns. The decode/analysis unit actually decodes the instructions in the buffer with a predefined scope of instruction examination, and it determines what dependencies exist between them and presents these results to the rules base and branch and pipeline analyzer. These subfunctions assess the "compoundability" of the incoming stream, and produce as output the recommended compounding or compoundings for the instructions in the buffer. The compounder then actually translates this information into compounded instructions with their tags.

When multiple compoundings are possible, the compounder can be designed to give preference to certain sequences of instructions, yielding more optimal compounding. The rules base via the categorization mechanism described earlier may contain rules for the complete instruction set or for a subset (as long as the subset is still a substantial part of the complete instruction-set architecture). It may additionally contain further information pertaining to the physical properties of the functional units, facilitating the embedding of control information in the tags. The rules base, though implementable in hard-wired, random logic, may also be implemented in some form of fast-access programmable storage, thereby allowing for flexibility as more functional units are added or removed, as more or fewer types of compoundings are desired, or even as the computing environment changes. For example, certain compoundings may be more advantageous in a commercial environment than in an engineering–scientific environment, or vice versa. If the rules base were programmable, such decisions could be made at machine configuration time. The structure of the CIC is similar to traditional cache designs,

with the addition of the tag bits. Thus, all traditional cache design techniques can also be applied to the CIC design. The actual incorporation of the tag bits into the cache line is an implementation issue and is not discussed further. Regarding the delay associated with the preprocessing, our System/370 implementation studies suggest that the delay through the ICU will be less than one machine cycle. Furthermore, our implementation studies suggest that this additional delay, depending on how cache misses are serviced, can be hidden in the cache miss service time.

Complex instruction set computer (CISC) architectures pose a number of challenges to the ICU/CIC mechanism. In brief, these challenges arise from variable-length instructions, data intermingled with instructions, and self-modifying code. It would seem that variable-length instructions could lead to portions of a CIC line being unanalyzed for compounding. Suppose an instruction fetch to address *A* has occurred and has caused a cache miss. Address *A* falls somewhere in a cache line other than the first location, 0. **Figure 5** illustrates the scenario. Straightforward left-to-right compounding leaves instructions in the range 0 to *A* unanalyzed, as depicted by the shaded region in the figure. This becomes a problem if an instruction fetch to the unanalyzed portion occurs. The problem is not one of determinism (i.e., the instructions can still be executed deterministically, as in any instruction cache), but is rather one of performance—namely, the instructions have not been compounded, and therefore may not be executed in parallel. Note that this problem does not arise in instruction-set architectures that have uniform instruction lengths, such as the 801 [21] or MIPS [22]. If instructions are always *p* bytes long, then, in the example, the ICU would know that an instruction begins at $A - p$, $A - 2p$, etc., and could analyze its way back to the beginning of the cache line.

One further complication can arise with any architecture, however: Unless cache lines are prefetched, compounding cannot occur across cache lines. For example, if two instructions can be compounded for parallel execution and a single instruction remains to be compounded at the end of a cache line, the last instruction cannot be compounded because the next instruction is unknown. The performance implications of this diminish as cache line size increases. For typical line sizes for high-performance systems, perhaps 64 bytes or greater, the impact is minimal, since the line likely contains many instructions that may have been successfully compounded. Prefetching the next sequential cache line can eliminate the problem because the next sequential instruction will be known and can thus be considered for compounding with the last instruction of the previous line. Additionally, the problem can be solved with compounding across lines that are already in the cache [23].

Unlike certain RISC architectures, CISC architectures

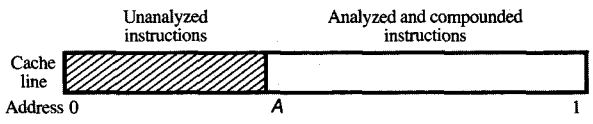


Figure 5

Unanalyzed instructions in a cache line.

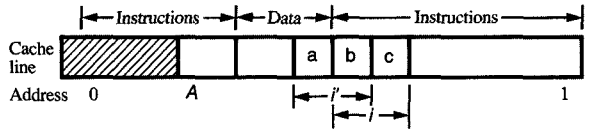


Figure 6

Instructions and data intermingled.

like System/370 may not require an instruction stream to be purely instructions. Data may reside anywhere between instructions. Consider the scenario illustrated in **Figure 6**, in which the ICU is instructed to begin analysis for compounding at address *A*. As it processes from *A* to the end of the cache line, it unknowingly compounds the data and loses its reference point to the true instruction boundaries, compounding datum *a* and instruction fragment *b* into an erroneous instruction *i'*, whereas the true instruction, *i*, is composed of instruction fragments *b* and *c*, and so on down the cache line. Later, a nonsequential instruction fetch occurs to instruction *i*. Consequently, the compoundings do not correspond to actual instructions, and are therefore invalid.

Note that, depending on the permissible lengths of instructions and data, not all architectures exhibit this problem, even when they allow data to be intermingled with instructions. Note also that the cache coherence protocol must account for the presence of data in the compound instruction cache when, for example, a store into a data region that is in the cache occurs. A discussion of cache coherence protocols is beyond the scope of this paper. The reader is referred to [24–26] for further consideration of these issues.

The solution to these problems is rooted in the degree to which the boundaries between instructions and data

(abstractly called *text*) can be exactly known. There are five distinct cases:

1. The text contains instructions only and the reference point, i.e., the boundary or address of the first instruction, is known.
2. The text contains instructions and data at known boundaries.
3. The text contains instructions only but the reference point is unknown.
4. The text contains instructions and data with partial reference points known.
5. The text contains instructions and data with unknown reference points.

The fifth case is the most general and worst-case scenario. Clearly, if it can be solved, both the unanalyzed portions of cache lines and data intermixed with instruction problems are solved, since both problems hinge on knowing a compounding reference point.[5] One algorithm that solves this case has the ICU examine each halfword (two bytes) in the line as though it were an instruction; i.e., each halfword is a potential instruction boundary.[6] Then, assuming an n-way compounding scheme, instructions are identified for a given halfword in the line by determining instruction lengths from that halfword and working to the right until n instructions have been identified. The compoundability of these instructions is then assessed, and a compounding tag is created for each halfword comprising the n instructions. This is repeated for every halfword in the line. When the process is complete, the tags for each halfword are merged to form the *tag vector* for the line (merging simply means selecting as the tag for a given halfword the tag with the highest value of all possible tags generated for that halfword). The validity of this algorithm can be established as follows. Consider a scheme that allows two-instruction compounding, as shown in **Figure 7**. Assume for simplicity that the rules for compounding indicate that 2- and 4-byte instructions are compoundable but 6-byte instructions are not. A single tag bit is required for each halfword: Let it be defined to be equal to one when the instruction starting with that halfword is compoundable with the next instruction to the right, and equal to zero when the instruction is not compoundable with the next instruction. In the figure it is assumed for simplicity that the text is examined serially. That is, a halfword is examined, and on the basis of its length (that for the example is assumed to be the only variable that determines compoundable instructions) a decision is reached as to whether or not it compounds with the next instruction; then the next halfword is considered and the process is repeated until the end of the text is reached. The text and the length associated with each halfword are depicted in part (a). In part (b), each line contains a pair of potential instructions on the left (TEXT), and the resulting compounding bits on the right (TEXT C-VECTOR). Moving down one line also advances the text to the next halfword until all halfwords have been examined and all possible C-vectors for the text have been produced. In part (c), these C-vectors are then reduced into a composite C-vector (CC-VECTOR), which is obtained by merging all of the C-vectors as described previously.

Note that only the first tag bit for an instruction in the C-vector is relevant to compounding; e.g., for a 6-byte instruction, there are three tag bits corresponding to the first, second, and third halfwords of the instruction, of which only the first bit is meaningful. Also note that because for any given halfword the maximum compounding may occur only when the halfword is examined, the construction of the CC-vector is immediate.

During program execution, instructions are fetched from the cache either sequentially or as branch targets, and correct program behavior is maintained by branching around data. Consequently, if an instruction is fetched sequentially, its tag indicates all instructions that are executable in parallel with that instruction and determines the beginning of the next sequential compound instruction. If any instruction in the compounded instructions is a branch, all instructions constituting the group following the branch are not executed if the branch is taken, but are executed if the branch is not taken. If an instruction is a branch target, its tag indicates the number of instructions following it that are executable in parallel and treated as the sequential instruction execution tag. If the text contains data, the data portion is not executed, because the branch preceding the data must be taken and the text following the branch is not executed. Note that because each halfword was examined as if it began an instruction, the tags can indicate more than one sequence of compound instructions depending on where the sequence begins. In the example, if the instruction beginning at byte 2 is accessed, either by sequential execution or as a branch target, the tag indicates that this instruction is compounded with the instruction at byte 4. Assuming no branches, the next instruction is the compound pair beginning at bytes 8 and 10. If, on the other hand, an execution sequence begins at byte 4, assuming for example that byte 4 is the target of a branch instruction, the tag indicates that instructions beginning at bytes 4 and 8 are executed in parallel, followed (assuming no branching) by the parallel execution of the instructions beginning at bytes 10 and 14. This indicates that even though compounding does not consider the dynamic instruction sequence when

[5] The solutions to the other cases can be found in [27, 28].

[6] In System/370, instructions are two, four, or six bytes long and must begin on halfword addresses. The first two bits of the first halfword of an instruction indicate its length.

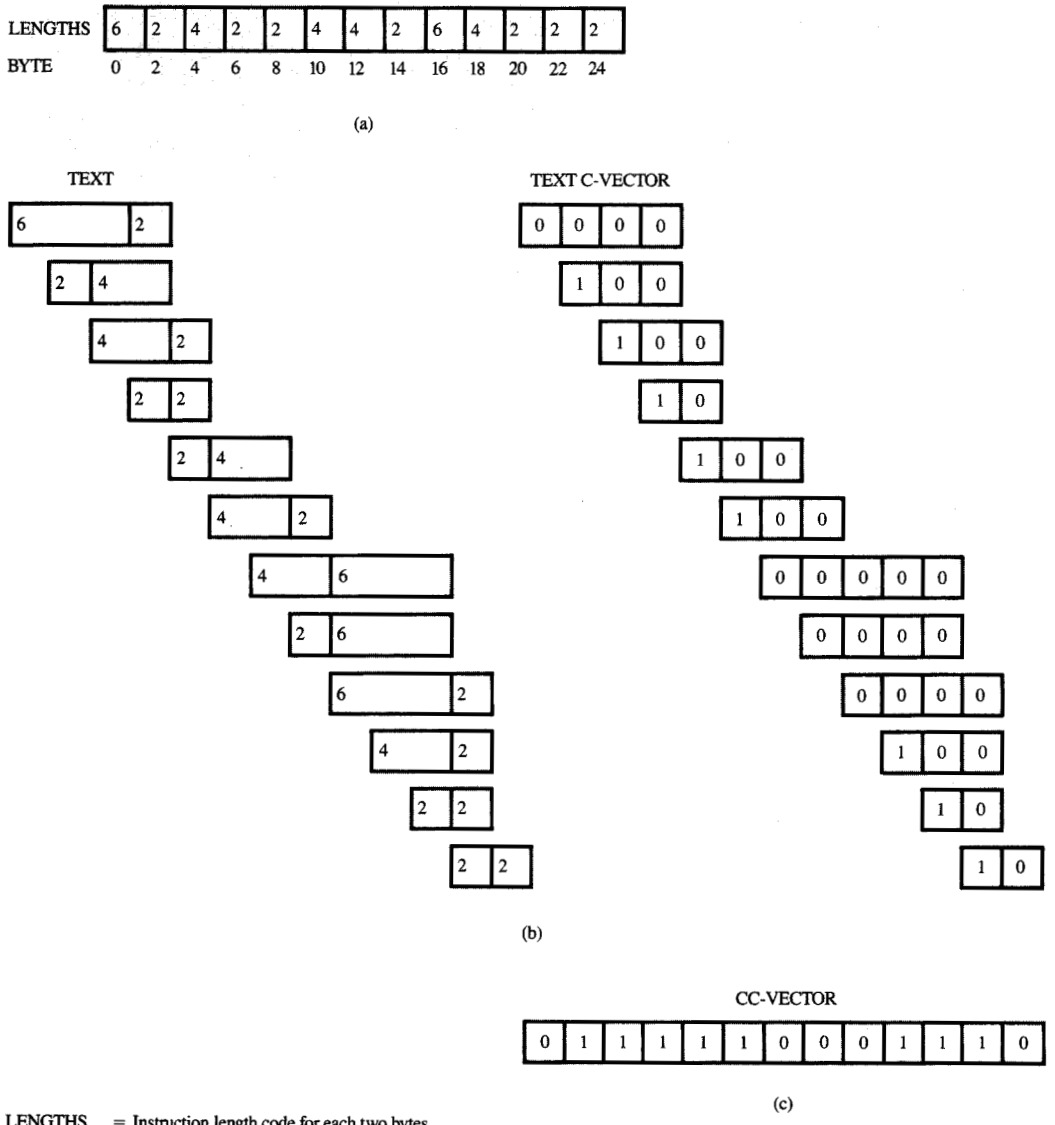


Figure 7

Worst-case compounding example.

determining the tags, the actual compound instructions executed are optimized for parallel execution for a given sequence of instructions. This algorithm has the further advantage of not requiring a reference point, which is helpful when compounding cache lines that have been rotated so that the instruction required by the instruction fetch hardware is received first. Indeed, even backward compounding can also be applied for this case. For System/370, backward compounding requires examination of the three halfwords preceding the halfword under examination to verify consistent instruction boundaries [28]. In any case, since the algorithm requires no reference points, no portions of a cache line go unanalyzed, and the presence of data in the line is irrelevant, since every halfword in the line is treated as though it were an instruction. Also note that although this algorithm has been

described in a sequential manner, there is nothing inherent in the algorithm that precludes a parallel implementation. One last idiosyncrasy, which few recently developed architectures allow, is self-modifying code. (Data areas that are frequently stored into and that are intermingled with instructions also exhibit this problem.) As with the other two architectural idiosyncrasies, a number of solutions offer various trade-offs between performance and complexity. Perhaps the most straightforward solution is this: The CIFC monitors stores from the functional units for stores into the instruction stream. When such a store occurs, instruction execution is halted, the stored-into CIC line is invalidated, and a fetch request for the modified line is issued to the memory subsystem. The remaining operations that must take place are identical to those that occur when a CIC miss occurs. While such a solution is straightforward, it assumes that stores to the instruction stream and/or data intermingled with instructions are infrequent. If this assumption does not hold true, other schemes have been developed that do not degrade performance when frequent stores are made into the instruction stream and/or text containing instructions is intermingled with data [29].

Earlier it was postulated that the performance-diminishing effects of branch-induced pipeline disruptions are even greater in superscalar processors than in scalar processors. It was suggested that even with branch prediction mechanisms in place, the disruptions may have a significant impact on performance. To counter these effects, the SCISM organization can employ a technique to completely overlap the execution of the most frequent kinds of branch instructions with other single or compound instructions. This technique, described in [30], is based on two principles, history-based branch prediction and early branch processing. The instruction-fetching unit processes instructions early with respect to the execution units. In so doing, it is able to detect a branch instruction in the incoming instruction stream, ascertain its predicted outcome, and then overlay it with the first instruction of the predicted stream, effectively removing the branch instruction from the stream. The overlaying technique is restricted to branch on condition (BC and BCR), the prevailing branch instructions in System/370, since these do not require a functional unit for their execution, but rather depend on the prior execution result, i.e., the condition code. It is then possible to completely overlap BC and BCR with other instructions, single or compound, that do require the execution units, provided that pipeline synchronizing hardware is used to ensure that the branch test is correlated with the proper execution results. A mechanism is used to block executions when a branch is mispredicted. A final important issue that requires further discussion is precise interrupt handling. To guarantee architectural compliance, provision must be made to present interrupts created during the execution of a compound instruction. For the SCISM machine configuration, the results must be the same as would be obtained had the members of the compound instruction been executed serially. It has been shown that interrupts for a compound instruction can be detected [14]. This is necessary but not sufficient: It is then required to attribute interrupts to the offending instruction. One means of doing this is to provide checkpointing hardware, where snapshots of the vital processor state are taken at predefined intervals, together with hardware to nullify the results of an entire compound instruction when an interrupt condition is detected (this same hardware can be used to block execution results when a branch instruction is mispredicted, as described above). The processor state may then be rolled back to the last known checkpoint and placed in an execution mode where all instructions are executed serially (i.e., parallel execution is disabled) until the interrupt resurfaces. At that point, it is clear which instruction caused the interrupt, and architectural compliance has been achieved. Given that interrupts are infrequent, this technique is attractive because of its implementation simplicity and also because it utilizes hardware that is already present in the implementation for other reasons, i.e., nullification hardware for mispredicted branches and checkpointing hardware for instruction retry. However, it is entirely possible to implement various other schemes for presenting architecturally precise interrupts; see for example [31–34].

Evaluation

The SCISM machine organization has been evaluated by trace-driven simulation using commercial workloads, since these are more representative of the typical machine execution environment than traditional benchmarks. However, results for Dhrystone [35] are also presented, since it seems to be a popular benchmark and is familiar to many. We simulated a two-way compounding scheme, maximum compounding of two instructions, to evaluate whether such a scheme holds promise for potential computer implementations. Other evaluations, including the parallelism increase and the architectural effects of RISC and CISC on parallelism using interlock collapsing ALUs, are reported elsewhere [36, 37]. Still other evaluations are entirely possible. In our evaluation, we measure the number of instructions that would actually be compounded in a hardware implementation of SCISM. The instruction-set architecture for purposes of evaluation was the IBM System/370 architecture. To avoid issues that clearly affect the performance of the superscalar machines but are entirely dependent on technology and implementation constraints, e.g., the number of instructions that can be paired and executed in parallel, modeled results are compared to the maximum

performance of a theoretical superscalar machine organization which issues and executes all instructions as pairs.

The evaluation consists of dividing the instruction set into categories and determining a set of compounding rules to use in compounding instructions between any two categories. Some assumptions on the processor hardware structure are needed to determine the compounding rules and categories. These are discussed in the section on hardware assumptions. A program was written to process the instruction trace by counting the number of compound instructions and the number of individual instructions actually executed. The compounding is performed as follows: Two instructions are examined and the rules are applied. If the instructions meet the rules, they are considered a compound instruction, and the following two instructions are then considered. If the instructions cannot be compounded, an attempt is made to compound the second with the next instruction. There are cases in which two instructions can be paired in hardware but the second is not executed; e.g., the first instruction is a taken branch. The second instruction, on the not-taken path, would be nullified in the CPU hardware. In processing the trace, the second instruction is never encountered, so the first executes singly and is counted as an individual instruction.

Since obtaining overall performance measurements (e.g., MIPS, cycles per instruction) depends greatly on the CPU implementation, including areas not related to parallel execution, the SCISM evaluation presented here is described in terms of that improvement which is due explicitly to parallel execution, independent of technology or implementation. Improvements due to parallel execution of instructions can be measured by the number of instructions which may execute in zero time, denoted by PZE (potential zero-cycle executions). The rationale behind this measurement is the following: If one instruction in a compound instruction pair executes in n cycles and the other instruction executes in $m \leq n$ cycles, the instruction taking m cycles to execute appears to execute in zero time. Because factors such as cache size and branch prediction accuracy vary from one implementation to the next, PZE measures the potential, not the actual, rate of zero-cycle execution. Additionally, note that zero-cycle instruction execution does not translate directly to cycles per instruction (CPI) because all instructions do not require the same number of cycles for their execution. The PZE measure simply indicates the number of instructions that potentially have been "removed" from the instruction stream during the execution of a program. For two-way compounding, at best half the instructions execute in zero time. Thus, for the theoretical superscalar machine the PZE is 50%, since it is assumed that every instruction is part of a pair. The PZE as defined thus far does not account for removal of branch instructions from the instruction stream as discussed earlier. When branches are removed from the instruction stream, they may also execute in zero time. When this is the case, PZE is defined to be the number of compound instruction pairs plus the number of branches removed from the instruction stream, all divided by the total number of instructions. In our evaluation, we consider two SCISM organizations: one that does not remove the branches from the instruction stream, denoted by SCISM1; and one that does, denoted by SCISM2.

Given that branches are not always predicted correctly, in order to include branch prediction success rate in the results of the evaluation we first determine the PZE of an instruction stream as if all the branches have been predicted correctly. Consequently, if the branch instruction is assumed to execute in zero time with correct prediction, we decrease the PZE for 100% successful prediction of branches by one for every incorrect branch prediction. The rationale for this choice is as follows:

- A branch that is neither compounded nor removed from execution is not included in the PZE for 100% prediction accuracy, and no adjustment need be made for misprediction. Note that in this case, while there is a performance degradation of the machine, the number of instructions that may execute as a pair (which is the concern of this evaluation) remains unchanged.
- A conditional branch removed from the execution stream takes zero cycles to execute if correctly predicted; however, if the prediction is incorrect, the branch stalls the pipeline and thus no longer executes in zero cycles.
- A compounded branch that is the first of a pair is originally included in the PZE. This branch must take the sequential path for it to have been compounded, since the evaluation is trace-driven. However, the branch may not be predicted correctly; in such a case, the processor would have attempted to execute the branch target rather than the sequential stream of the trace, implying that the sequential instruction requires execution cycles, i.e., not a zero-cycle execution for the pair.
- A compounded branch that is the second of a pair is also originally included in the PZE. It is debatable in this case whether the PZE should be adjusted. On one hand, the execution of the first instruction in the pair is overlapped by the branch, regardless of the prediction, and it can be considered to execute in zero cycles. On the other hand, the total time used for the execution of this compound pair is increased by a wrong prediction. Consequently, depending on which instruction is charged for the pipeline stall, whether or not the PZE is adjusted is a matter of charging either the parallel execution of the instructions or the execution of the single branch

Table 1 IBM System/370 instruction set divided into categories.

Category	*Description*	*Instructions*	*Resources used*
1	RR-format loads, logicals, arithmetics, compares	LCR, LPR, LNR, LR, LTR, NR, OR, XR, AR, SR, ALR, SLR, CLR, CR	ALU
2	RS-format shifts	SRL, SLL, SRA, SLA, SRDL, SLDL, SRDA, SLDA	Shifter
3	Branches on count and index	BCT, BCTR, BXH, BXLE	BU, ALU
4	Branches on condition	BC, BCR	BU
5	Branches and link	BAL, BALR, BAS, BASR, BSM, BASSM	BU, ALU
6	Stores	ST, STH, STC, STCM, MVI, STD, STE	AU, DC, shifter
7	Loads	L, LH, LD, LE	AU, DC
8	Load address	LA	AU
9	RX RS SI-format logicals, arithmetics, inserts, compares	N, O, X, A, AH, AL, S, SH, SL, IC, ICM, C, CH, CL, CLI, CLM	AU, DC, ALU
10	Test under mask	TM	AU, DC, ALU
11	Control (no storage reference)	LRA, SPM, SPKA, IPK	Various
12	RR-format floating-point	AER, ADR, AXR, AUR, AWR, CER, CDR, DER, DDR, DXR, HER, HD, LER, LDR, LTER, LTDR, LCER, LCDR, LNER, LNDR, LPER, LPDR, LRER, LRDR, MER, MDR, MXR, MXDR, SER, SDR, SXR, SUR, SWR, DR, MR	FPU
13	RX-format floating-point	AD, AE, AW, AU, CD, CE, DD, DE, MD, ME, SD, SE, SW, SU, D, M, MH	AU, DC, FPU
14	Miscellaneous one-storage block access	NI, OI, XI, LM, STM, CVB, CVD, EX, STNSM, STOSM, TPROT	AU, ALU, shifter, DC
15	Miscellaneous two-storage block access	NC, OC, XC, CLC, CLCL, MVC, MVCL, MVCIN, MVN, MVO, MVZ, MVCK, PACK, UNPK, TR, TRT, AP, CP, DP, ED, EDMK, MP, SRP, SP, ZAP	AU, ALU, shifter, DC
16	All other instructions	Various	Various

instruction. We chose to charge the parallel execution rather than the branch execution and to decrease the PZE count, because it reflects more accurately the actual zero-cycle execution of instructions in a program, and leads to a uniform treatment of incorrect branch predictions.

- *Hardware assumptions*

The definitions for categorization of instructions suggest that the CPU hardware in an implementation determines the categorization of instructions and the rules for compounding. The modeled SCISM implementation consists of the following hardware assumptions:

- One two-input ALU.
- One three-input ALU for dependency collapsing.
- Two shifters.
- One three-input address unit (AU).
- One four-input address unit for dependency collapsing.
- One two-port data cache (DC).
- One floating-point unit (FPU).
- One branch unit (BU).

Clearly, in an actual implementation, cost–performance trade-offs could be made by eliminating functions. For example, if it is determined that the incidence of compound shift instructions is rare, the second shifter

Table 2 Compounding rules. Two-port cache, five-stage pipeline.

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*
1	Y	A	Y	Y	Y	A	Y	Y	Y	Y	A	I	Y	Y	Y	Y
2	I	I	I	I	I	I	I	I	E	I	I	I	E	E	E	E
3	Y	A	N	N	N	A	Y	Y	Y	Y	A	I	Y	Y	Y	Y
4	Y	Y	N	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
5	Y	A	N	N	N	A	Y	Y	Y	Y	A	Y	Y	Y	Y	Y
6	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
7	I	I	I	I	I	I	I	I	E	I	I	I	E	E	E	E
8	Y	Y	E	I	E	E	I	I	E	Y	E	I	E	E	Y	E
9	Y	I	E	I	E	I	I	I	E	I	I	I	I	I	I	I
10	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
11	I	I	I	I	I	I	I	I	I	I	N	N	N	N	N	N
12	I	I	I	I	I	I	I	I	I	I	N	N	N	N	N	N
13	I	I	I	I	I	I	I	I	I	I	N	N	N	N	N	N
14	I	I	I	I	I	I	I	I	I	I	N	N	N	N	N	N
15	I	I	I	I	I	I	I	I	I	I	N	N	N	N	N	N
16	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

Legend:
Y = yes, always compound
N = no, never compound
I = compound only if independent (no dependency)
E = compound if execution dependency or independent
A = compound if address-generation dependency or independent

could be eliminated with little loss in performance. Note that since the focus of this study is on commercial workloads, there is no attempt to execute two floating-point instructions simultaneously, as indicated by the presence of one non-interlock-collapsing floating-point unit. In addition to the units listed, the CPU contains a microcode controller and storage for the execution of complex instructions. A separate instruction cache may be advisable for performance, since there can be instances of two data accesses at the same time instructions are being fetched. This choice affects the hardware design but is not relevant for the compounding measurements in this study. Problematic circumstances generated by a member of a compound instruction during instruction execution, e.g., store–load interlock in the two-port data cache, are assumed to be handled by hardware and do not affect compounding.

Obviously, the definition of the categories for SCISM, as well as the particular compounding rules, depends on the pipeline structure. For this study, we assume that the CPU pipeline for single-execution-cycle instructions consists of five stages: instruction fetch, instruction decode, address generation, execution, and register put-away. For load and store instructions, the cache access occurs in the execution stage using the address determined in the preceding stage. The memory address is computed from the addition of two registers and a twelve-bit displacement in the address generation cycle. An RR-format in System/370, for example the add instruction, adds the content of one register to the value in a second register during the execution stage. A number of instructions require both a cache access and a computation. An RX-format add, for example, adds a four-byte value from memory to a register. In this case, an extra execution stage is added to the pipeline. Complex instructions may use multiple execution stages, under control of microcode, which can use a variety of functional units.

The System/370 instruction set was divided into 16 categories based on the hardware utilization of each instruction. The 16 categories are listed in **Table 1**. Refer to [6] for a detailed explanation of each of these instructions. The conditional branches, Category 4, are considered for removal from the execution pipeline because they do not require use of execution-type hardware (ALU). Other branches have execution cycles and update registers in addition to performing a branch and therefore remain in the compound instruction stream.

On the basis of the hardware assumptions, a set of compounding rules was created which would be implemented in the ICU. A summary of the rules appears in **Table 2**. The table is read by finding the category of the first instruction and reading across the row to the category of the second instruction. The entry in the table indicates whether the two instructions always compound (because they have no dependency or because any dependency can be collapsed); never compound (due to conflicting resource requirements); compound only if there is no dependency; can collapse an execution dependency but not an address dependency; or can collapse an address dependency but not an execution dependency. For example, the table indicates that Category 8 (load address) and Category 7 (load) instructions must be independent. This is because collapsing an address generation dependency would require

Table 3 Characteristics of instruction traces.

	TSO		*IMS*		*CICS*		*VM*		*RAMPC*		*Dhrystone*	
Instructions	1363137		1349345		1362623		3404680		1361600		3735	
Dependencies (%)	29.2		26.9		30.4		21.0		18.8		27.7	
Category	*Instruction frequency and rank by category*											
1	13.7	3	13.8	3	12.2	4	10.4	4	10.4	4	18.7	1
2	1.2	13	1.0	13	0.8	13	1.8	12	1.1	11	0.8	9
3	2.7	10	1.1	12	0.6	14	3.0	10	0.4	13	0.3	10
4	21.2	1	21.5	1	22.1	1	25.4	1	28.8	1	18.2	2
5	2.5	11	2.8	10	2.7	10	2.7	11	2.7	10	4.8	7
6	9.1	5	9.0	5	8.9	5	7.3	6	6.4	7	10.4	5
7	15.9	2	18.2	2	17.0	2	13.2	2	11.4	2	9.9	6
8	4.8	9	5.4	8	5.1	7	8.1	5	5.8	8	18.1	3
9	11.7	4	9.5	4	12.8	3	10.9	3	10.6	3	16.6	4
10	5.7	6	6.0	6	5.7	6	6.1	7	10.1	5	0.0	14
11	0.4	14	0.3	15	0.3	15	0.1	15	0.1	16	0.0	14
12	0.1	16	0.1	16	0.1	16	0.1	16	0.1	15	0.1	12
13	0.2	15	0.5	14	1.3	11	0.3	14	0.2	14	0.3	10
14	5.5	7	5.8	7	4.7	8	5.9	8	3.9	9	1.6	8
15	4.2	8	3.7	9	4.6	9	4.0	9	7.2	6	0.1	12
16	1.3	12	1.4	11	1.1	12	0.7	13	0.9	12	0.0	14

a five-input adder for the load instruction,[7] which conflicts with the hardware assumptions. Note that no execution dependency can exist between these two instructions because the load instruction reads registers only for address generation, not for execution.

When the first of a potential pair requires more pipeline stages than the second instruction (RX-format followed by an instruction from one of several categories), the pipeline of the second instruction may have been extended so that the execution of the second instruction does not precede the execution of the first instruction. This is not the case if the Category 9 instruction is second in a pair. When SCISM2 is considered, i.e., when conditional branches are removed from the instruction stream, all Category 4 rules indicate no compounding.

To further understand Table 2, consider a few examples. Consider compounding two Category 1 instructions. Since both require an ALU and any dependency could be collapsed on the available hardware, these instructions can always be compounded [except for the degenerate case (see footnote 3) where the instructions are not compounded and are counted by the model as instructions executed serially]. Two Category 2 instructions can compound only if independent, because a shifter cannot collapse a dependency with another unit. Category 1 and Category 2 could compound if there is a dependency in address generation (calculation of shift amount), but not if the dependency requires a compound add–shift. Category 2 and Category 1 can compound only if independent.

[7] Note that it may be possible for the ICU to detect whether one or more address inputs is zero in the two instructions, allowing the address dependency to be collapsed with a four-input adder. However, this hardware capability was not assumed, and the interlock cannot be collapsed.

Instructions in Categories 11–16 typically require multiple cycles for execution. It is assumed, therefore, that they cannot be compounded with one another and that they can be first of a pair only if independent of the second instruction. Category 16 instructions can never be first in a pair because many of these instructions change the control state of the processor.

- *Instruction traces and results*

The basis for the measurement of programs comes from the instruction traces of several workloads. Each trace record gives the contents of the instructions, the addresses of instructions and operands, and the contents of operands. Compounding is determined first by observing whether the two instructions are adjacent in memory, and second by examining the instructions themselves, determining what dependencies exist among the instructions, and applying the compounding rules.

We used the following traces:

- TSO: representative workload.
- IMS: hierarchical database running on MVS.
- CICS: transaction processing running on MVS.
- VM: interactive user workload.
- RAMP-C: transaction processing benchmark.
- Dhrystone: synthetic benchmark [35], PL.8 compiler.

Table 3 lists the traces and some of their characteristics. For each trace, the table shows the number of dynamically consecutive instructions that have a register dependency. The frequency of each instruction category is also listed.

Table 4 Potential zero-cycle execution instructions (%). SCISM PZE and fraction of theoretical best superscalar PZE for various branch-prediction rates.

Benchmark	*Theoretical PZE*	*SCISM1 PZE*	*SCISM2 PZE*	*SCISM1 ÷ Theoretical*	*SCISM2 ÷ Theoretical*	*Branch prediction accuracy*
TSO	50.0	39.9	47.4	0.798	0.948	100
IMS	50.0	39.7	48.2	0.794	0.964	100
CICS	50.0	40.1	47.6	0.802	0.952	100
VM	50.0	38.9	48.2	0.779	0.964	100
RAMPC	50.0	40.4	49.0	0.808	0.980	100
Dhrystone	50.0	43.6	47.7	0.872	0.954	100
TSO	48.7	39.0	46.2	0.801	0.948	95
IMS	48.7	38.8	47.1	0.797	0.966	95
CICS	48.7	39.2	46.5	0.805	0.954	95
VM	48.4	38.0	46.8	0.784	0.966	95
RAMPC	48.4	39.2	47.5	0.810	0.981	95
Dhrystone	48.8	42.7	46.7	0.875	0.956	95
TSO	46.0	37.2	43.7	0.808	0.950	85
IMS	46.2	37.1	44.7	0.803	0.967	85
CICS	46.2	37.4	44.0	0.809	0.952	85
VM	45.3	36.1	44.0	0.796	0.971	85
RAMPC	45.2	36.9	44.4	0.816	0.982	85
Dhrystone	46.5	40.9	44.6	0.879	0.959	85
TSO	43.4	35.4	41.3	0.815	0.952	75
IMS	43.6	35.4	42.3	0.811	0.969	75
CICS	43.6	35.5	41.5	0.814	0.952	75
VM	42.2	34.2	41.2	0.809	0.976	75
RAMPC	42.0	34.6	41.3	0.823	0.984	75
Dhrystone	44.2	39.2	42.5	0.887	0.962	75

(Since Dhrystone consists of repetitions of the same large loop, only a few iterations were used in the measurements.)

The compounding of instructions was simulated for the traces. Compounding data were gathered for SCISM using the compounding rules of Table 2. The results in **Table 4** show the PZE for each trace when Category 4 branches are compoundable, denoted "SCISM1," and when they are removed from execution, denoted "SCISM2." The table shows the PZE relative to a theoretical superscalar computer. The effect of branch prediction accuracy on PZE is shown for each case.

Several observations can be made from Table 4. First, it can be noted that SCISM1 achieves roughly 78–88% of the theoretical maximum performance. For SCISM2, 94–98% of the theoretical maximum PZE is achieved. The three MVS traces tend to be similar in the degree of parallel execution of instructions.

Differences between traces can be understood by examining Table 3. Several characteristics of programs are important in determining the degree of compounding success. Simpler instructions, where dependencies can be collapsed, and instructions that do not modify registers tend to be more "compoundable" than other instructions. Categories 1, 6, and 10 are examples of this. Referring to the 100% prediction accuracy portion of the table, the following can be verified: for SCISM1, Dhrystone has the highest frequency of these three categories and has the highest PZE (43.5), while VM has the lowest frequency and the lowest PZE (38.9). While Table 3 does not show frequencies of pairs of instructions or categories, there are some well-known pairs in commercial System/370 programs. One example is the instruction sequence TM followed by BC. This pair is always compoundable for SCISM1, according to the rules in Table 2. RAMP-C has the highest frequency of Category 10 (TM), and this contributes to the high PZE for the trace. The frequency of branches also contributes to the PZE. With more branches, there are fewer nonbranching instructions between branches. If those branches are taken, the lengths of sequences of nonbranching instructions are shorter, resulting in less compounding potential. Fewer branches mean longer sequences not interrupted by a branch. RAMP-C and VM have a high frequency of branches, while Dhrystone has a low frequency. Finally, although SCISM collapses register dependencies between instructions, Table 2 indicates that this is not possible for any arbitrary pair on the assumed hardware. RAMP-C has a low number of dependencies, and CICS has a high number. Thus, one expects more compounding in RAMP-C than in CICS. While predicting the exact PZE from the statistics may not be feasible, the statistics do give some

indication of which programs lead to better results on a SCISM processor.

The same statistics are important for studying the SCISM2 results. Here, however, the frequency of Category 4 branches affects results differently. Because of the removal of a percentage of instructions, the compounding is reduced, but because of the zero-cycle execution of branches, the PZE is increased. When branches are not compoundable, those traces with a higher frequency of branches do better. All of these removed branches contribute to the PZE; when branches are compoundable, those compounded contribute to the PZE, while those not compounded are not part of the PZE. VM, with a large number of branches, has the highest PZE, while Dhrystone has the fewest branches, and nearly the lowest PZE. This is a reversal of the results for SCISM1. On the other hand, RAMP-C has a high PZE for both SCISM1 and SCISM2. Perhaps conditional branches compound frequently in SCISM1 because of the large number of TM instructions. In SCISM2, those branches are not compounded, but are removed from the instruction stream.

As the branch-prediction parameter is set to lower values, those traces with lower frequencies of branches perform better than other traces. For example, Dhrystone has the fourth highest PZE for SCISM2 and 100% prediction, but the second highest PZE for SCISM2 and 75% prediction. The relationships between the columns also change. The SCISM1 PZE decreases more slowly than the SCISM2 PZE; for 75% prediction, the two are closer in PZE than they were for 100% prediction. This is because in SCISM2 all Category 4 branches are part of the PZE and thus are subject to branch-prediction error penalties. In SCISM1, some are not part of the PZE; thus, an incorrect prediction does not affect the PZE. Also of interest is the performance of SCISM compared to the theoretical superscalar. Both SCISM1 and SCISM2 have better relative performance as prediction accuracy decreases. This is indicative of the different ways in which the two schemes handle branches. For superscalar processors, correct prediction is necessary to pair a branch with its target. For SCISM, branch performance is achieved through static compounding and removal from the instruction stream.

SCISM1 provides a significant portion of the theoretical instruction-level parallelism in commercial programs. By removing conditional branches from the instruction stream, SCISM2 increases the relative PZE by about 8–23% over SCISM1. For realistic branch prediction, say 85%, SCISM2 is roughly 95–98% and SCISM1 is roughly 79–87% of the best superscalar PZE. For all traces, SCISM2 exceeds 95% and SCISM1 exceeds 77% parallel execution of instructions relative to the theoretical superscalar machine.

Conclusions

On the basis of the assumption that no single mechanism provides a significant performance enhancement in von Neumann instruction-level parallel processors, we have identified a number of problems that inhibit parallel execution, and have presented a novel machine organization, the scalable compound instruction set machine (SCISM), which incorporates a number of mechanisms that either solve or alleviate superscalar machine deficiencies. In summary, SCISM comprises the following concepts and mechanisms:

1. Instruction categorization by hardware utilization rather than opcode description, which provides the capability for improving the number of possible groupings of instructions for parallel execution while avoiding prohibitive hardware increases and loss of performance due to increased cycle time and additional pipeline stages in an implementation.
2. Interlock collapsing hardware that provides the capability of parallel execution of interlocked instructions.
3. A broader scope of instruction text examination than instruction stack text examination, which provides the capability for improving the degree of instruction-level parallelism extracted from a program, and avoids prohibitive hardware requirements, increased cycle time, and FIFO preprocessing.
4. Mechanisms that avoid on-demand preprocessing of the instruction text, maintain the preprocessing "permanently," and perform "static" preprocessing in the cache.
5. A reduction of the amount of logic required for parallel instruction fetch/issue/decode in the instruction stack.
6. Incorporation of tagging or decoding mechanisms in the cache that can facilitate the parallel issue/execution of instructions, out-of-order execution, decoding of instructions, branching, functional unit scheduling, routing of operands to the functional units, and so forth.
7. Handling of architectural idiosyncrasies with negligible loss of performance and avoidance of prohibitive hardware requirements.
8. A flexible location for the preprocessing facility, allowing implementation constraints and performance/cost trade-offs to be accommodated.

The evaluation of the mechanisms using an instruction-level parallel-execution machine capable of executing at most two instructions against a theoretical superscalar machine suggests that with an efficient branch handling mechanism the proposed machine organization can accomplish parallel execution of instructions that exceed the 90% range of the theoretical machine. This is accomplished despite instruction dependencies and with

less complexity than comparable traditional superscalar designs, suggesting that SCISM represents a promising new computer organization.

Regarding future research for SCISM or SCISM-like processors, it may be of interest to explore the possibility of expanding the complex CISC instructions to RISC-like primitives and then possibly combining the RISC primitives in compound instructions in hardware, using an approach similar to that described by Silberman and Ebcioglu in [38].

Acknowledgments

The authors thank T. Jeremiah, J. Phillips, and W. Kelley for valuable comments. Additionally, Stamatis Vassiliadis thanks C. Conti, N. King, J. Roossien, and E. Shimp for the support and encouragement they provided for the development of the SCISM concepts.

References

1. N. P. Jouppi, "The Nonuniform Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance," *IEEE Trans. Computers* **38,** 1645–1658 (December 1989).
2. R. W. Horst, R. L. Harris, and R. L. Jardine, "Multiple Instruction Issue in the NonStop Cyclone Processor," *Proceedings of the 17th International Symposium on Computer Architecture*, IEEE Computer Society, Los Alamitos, CA, May 1990, pp. 216–226.
3. *IBM Journal of Research and Development*, "The IBM RISC System/6000 Processor," Special Issue, Vol. 34, No. 1, January 1990.
4. N. P. Jouppi and D. W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," *Proceedings of ASPLOS III*, ACM, 1989, pp. 272–282.
5. J. E. Smith, "Dynamic Instruction Scheduling and the Astronautics ZS-1," *IEEE Computer* **27,** 21–35 (July 1989).
6. *IBM Enterprise System Architecture/370 Principles of Operation*, Order No. SA22-7200-0, 1989; available through IBM branch offices.
7. A. Padegs, B. B. Moore, R. M. Smith, and W. Buchholz, "The IBM System/370 Vector Architecture: Design Considerations," *IEEE Trans. Computers* **37,** No. 5, 509–520 (May 1988).
8. H. S. Warren, "Instruction Scheduling for the IBM RISC System/6000 Processor," *IBM J. Res. Develop.* **34,** No. 1, 85–92 (January 1990).
9. J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer* **17,** No. 1, 6–22 (January 1984).
10. J. E. Smith, "A Study of Branch Prediction Strategies," *Proceedings of the 8th Annual Symposium on Computer Architecture*, IEEE Computer Society, Los Alamitos, CA, May 1981.
11. M. Putrino and S. Vassiliadis, "Resolution of Branching with Prediction," *Int. J. Electron.* **66,** No. 2, 163–172 (February 1989).
12. S. Vassiliadis and M. Putrino, "Condition Code Predictor for Fixed-Point Arithmetic Units," *Int. J. Electron.* **66,** No. 6, 887–890 (June 1989).
13. S. Vassiliadis, J. E. Phillips, and B. Blaner, "Interlock Collapsing ALUs," *IEEE Trans. Computers* **42,** 825–839 (July 1993).
14. S. Vassiliadis, J. E. Phillips, and B. Blaner, "ICU Design Considerations," *Technical Report TR-01.C114*, IBM Glendale Laboratory, Endicott, NY, October 1991, p. 22.
15. S. Vassiliadis and J. E. Phillips, "Interlock Collapsing ALU Design," *Technical Report TR-01.C115*, IBM Glendale Laboratory, Endicott, NY, October 1991, p. 37.
16. J. E. Phillips and S. Vassiliadis, "High Performance 3-1 Interlock Collapsing ALUs," *IEEE Trans. Computers,* accepted for publication.
17. R. J. Eickemeyer and S. Vassiliadis, "A Load Instruction Unit for Pipelined Processors," *IBM J. Res. Develop.* **37,** 547–564 (July 1993).
18. S. Vassiliadis, B. Blaner, R. J. Eickemeyer, J. Phillips, and N. Malik, "In-Cache Pre-Processing and Decode Mechanisms for Fine-Grain Parallelism in SCISM," *Proceedings of the IEEE Phoenix Conference on Computers and Communication*, Phoenix, AZ, March 1993, pp. 91–97.
19. S. Vassiliadis and B. Blaner, "Concepts of the SCISM Organization," *Technical Report TR-01.C209*, IBM Glendale Laboratory, Endicott, NY, January 1992, p. 14.
20. R. J. Eickemeyer, S. Vassiliadis, and B. Blaner, "An In-Memory Preprocessor for SCISM Instruction-Level Parallel Processors," *Technical Report TR-01.C407*, IBM Glendale Laboratory, Endicott, NY, May 1992, p. 16.
21. G. Radin, "The 801 Minicomputer," *IBM J. Res. Develop.* **27,** No. 3, 237–246 (May 1983).
22. J. Hennessy, N. Jouppi, F. Baskett, and J. Gill, "MIPS: A VLSI Processor Architecture," *Proceedings of the CMU Conference on VLSI Systems and Computations*, Carnegie Mellon University, Pittsburgh, PA, 1981.
23. B. Blaner and T. L. Jeremiah, "Cross-Cache-Line Compounding Algorithm for SCISM Processors," patent pending, IBM U.S. Docket No. EN991129, 1992.
24. A. J. Smith, "Cache Memories," *Computing Surv.* **14,** No. 3, 473–530 (September 1982).
25. J. Archibald and J. L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. Computer Syst.* **4,** No. 4, 273–298 (November 1986).
26. J. L. Baer and W. H. Wang, "Multilevel Cache Hierarchies: Organizations, Protocols, and Performance," *J. Parallel & Dist. Computing* **6,** 451–476 (1989).
27. B. Blaner and S. Vassiliadis, "A Hardware Preprocessor for Instruction-Level Parallel Processors," *Technical Report TR-01.C208*, IBM Glendale Laboratory, Endicott, NY, January 1992, p. 13.
28. R. J. Eickemeyer and S. Vassiliadis, "Compounding Algorithms for SCISM Processors," *Technical Report TR-01.C404*, IBM Glendale Laboratory, Endicott, NY, May 1992, p. 18.
29. R. J. Eickemeyer, S. Vassiliadis, and B. Blaner, "Memory Management for Scalable Compound Instruction Set Machines with in Memory Compounding," U.S. Patent 5,197,135, 1990.
30. B. Blaner, T. L. Jeremiah, and S. Vassiliadis, "A Branch Instruction Processor for SCISM Architectures," *Technical Report TR-01.C437*, IBM Glendale Laboratory, Endicott, NY, May 1992, p. 20.
31. S. G. Tucker, "The IBM 3090 System: An Overview," *IBM Syst. J.* **25,** No. 1, 4–19 (1986).
32. A. Y. Ngai and B. Blaner, "Apparatus and Method for Implementing Precise Interrupts on a Pipelined Processor with Multiple Functional Units with Separate Address Translation Means," U.S. Patent 5,003,462, March 1991.
33. W. W. Hwu and Y. N. Patt, "Checkpoint Repair for High-Performance Out-of-Order Execution Machines," *IEEE Trans. Computers* **C-36,** 1496–1514 (December 1987).

34. H. C. Torng and M. Day, "Interrupt Handling for Out-of-Order Execution Processors," *IEEE Trans. Computers* **42,** No. 1, 122–126 (January 1993).
35. R. P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Commun. ACM* **27,** No. 10, 1013–1030 (October 1984).
36. N. Malik, R. J. Eickemeyer, and S. Vassiliadis, "Interlock Collapsing ALU for Increasing Instruction Level Parallelism," *Conference Proceedings of the 25th Annual International Symposium on Microarchitecture*, Portland, OR, December 1992, pp. 149–157.
37. N. Malik, R. J. Eickemeyer, and S. Vassiliadis, "Architectural Effect on Dual Instruction Issue with Interlock Collapsing ALUs," *Conference Proceedings of the IEEE Phoenix Conference on Computers and Communication*, Phoenix, AZ, March 1993, pp. 42–48.
38. G. M. Silberman and K. Ebcioglu, "An Architectural Framework for Supporting Heterogeneous Instruction-Set Architectures," *IEEE Computer*, pp. 39–56 (June 1993).

Received March 1, 1993; accepted for publication January 6, 1994

Stamatis Vassiliadis *IBM RISC System/6000 Division, 11400 Burnet Road, Austin, Texas 78758 (STAMOS at AUSVM6, stamatis@vnet.ibm.com).* Dr. Vassiliadis received the Dr.Eng. degree in electronic engineering from the Politècnico di Milano, Milan, Italy, in 1978. He is currently employed at the Advanced Workstations and Systems Laboratory in Austin, Texas, after previous assignments at the Mid-Hudson Valley Laboratory, IBM Poughkeepsie, and the Glendale Laboratory, IBM Endicott, New York. His work assignments include the development of new computer organizations and architectures, high-level design and technical leadership in the implementation of new computer systems, and advanced research in a variety of computer-related fields. Previous work included participation in the design of the IBM 9370 Model 60 computer system. Since joining IBM he has received a number of awards, including 18 levels of the IBM Publication Achievement Award, 13 levels of the IBM Invention Achievement Award, and an IBM Outstanding Innovation Award for Engineering/Scientific Hardware Design in 1989. In 1990 he was awarded the most patents in IBM. His research interests include computer architecture, hardware design and functional testing of computer systems, parallel processors, computer arithmetic, EDFI for hardware implementations, neural networks, fuzzy logic and systems, and software engineering. Dr. Vassiliadis has been an Adjunct Professor and a Visiting Professor in the School of Electrical Engineering, College of Engineering, Cornell University, Ithaca, New York, and in the Electrical Engineering Department at the Thomas J. Watson School of Engineering and Applied Science, State University of New York (S.U.N.Y.), Binghamton, New York.

Bart Blaner *IBM Microelectronics Division, Burlington facility, Essex Junction, Vermont 05452 (BLANER at BTVLABVM, blaner@vnet.ibm.com).* Mr. Blaner is an Advisory Engineer in the Microprocessor and Mwave Product Development group at the IBM Burlington Laboratory. His current assignment is digital signal processor design. Previously he worked in the IBM Glendale Laboratory, Endicott, New York, where he contributed to a variety of processor projects ranging from dataflow computer research to high-performance mainframe computer development. His technical interests include processor design and implementation, computer architecture, VLSI design, and computer networks. Mr. Blaner received the B.S. degree in electrical and computer engineering, with great distinction, from Clarkson University in 1983. He holds five patents, has attained four levels of the IBM Invention Achievement Award, and is a member of Tau Beta Pi, Eta Kappa Nu, and Phi Kappa Phi.

Richard J. Eickemeyer *IBM Application Business Systems, 3605 Hwy. 52 North, Rochester, Minnesota 55901 (RJE at RCHVMV3, rje@vnet.ibm.com).* Dr. Eickemeyer is an Advisory Engineer in the IBM Rochester Laboratory Hardware Design Center. His current assignment is performance analysis of AS/400® processors. Prior to moving to Rochester, he worked in the IBM Glendale Laboratory, Endicott, New York, on system design and processor performance. Dr. Eickemeyer received the B.S. degree in electrical engineering from Purdue University and the M.S. and Ph.D. degrees from the University of Illinois at Urbana-Champaign. His research interests are computer architecture, parallel processing, and performance analysis. Since joining IBM he has received awards which include two levels of the IBM Publication Achievement Award and two levels of the IBM Invention Achievement Award.