FUTURE SUPERSCALAR PROCESSORS BASED ON INSTRUCTION COMPOUNDING

J. E. Smith

University of Wisconsin-Madison
Email: jes@ece.wisc.edu

ABSTRACT

Future processor cores will achieve both high performance and power efficiency by using relatively simple hardware designs. This paper proposes and describes such a future processor in some detail, to illustrate the key features of future designs. A key feature of the future processor is the use of compounded, or fused, instructions, as first proposed by Stamatis Vassiliadis. A second key feature of the future processor is an integrated, hard-ware/software co-designed implementation; the software component shifts complexity away from hardware by performing transparent translation and optimization of machine code. A third key feature of the future processor is a dual decoder front-end that dramatically reduces start-up delays found in most software-translation implementations. Because of its dominance in the marketplace, the x86 ISA is employed for describing and evaluating the future processor.

1. INTRODUCTION

Future processor cores will be engineered to achieve a fine balance between performance and power efficiency. Instruction compounding, or fusing[1], a technique pioneered by Stamatis Vassiliadis, is currently used in leading-edge microprocessors. When supported by advanced binary translation mechanisms, both software and hardware, instruction compounding will be an important feature of future high performance, high efficiency processors. It is the goal of this paper to chart a direction for future superscalar processors by describing one potential design, simply referred to as the "future processor", which incorporates instruction compounding, transparent binary translation and optimization, and a dual-decoder frontend.

In the future processor, instruction compounding mechanisms produce multi-operation instructions by first de-composing complex instructions into simpler operations or *micro-ops* (as with a CISC instruction set), or by starting with simple instructions (as with a RISC instruction set). Then the micro-ops are re-arranged and fused into compound instructions or *macro-ops*. Compounding has a number of important advantages: 1) it reduces the number of instructions that must be processed by the execution pipeline, and this gives higher execution throughput for a given superscalar processor width; 2) it makes some register data dependences explicit, thereby enabling the effective use of collapsed 3-1 ALUs; 3) when the compounding algorithm incorporates a heuristic that selects a single-cycle ALU micro-op as the first of a fused macro-op pair, then single cycle register-register instructions can be largely eliminated, thereby simplifying instruction issue and data forwarding logic.

In the IBM SCISM project (Scalable Compound Instruction Set Machine) [1], Stamatis Vassiliadis and his colleagues targeted a CISC ISA – the IBM 360/370. Today, the dominant ISA for general purpose computing is also CISC - the Intel x86 ISA. Furthermore, it is likely to remain the dominant ISA for at least the next decade, probably much longer. Consequently, it is natural that high performance and efficient CISC processor designs, and the x86 in particular, should be the focus of microarchitecture research. The x86 ISA is used as the vehicle for describing the future processor proposed in this paper; the extension to other ISAs should be relatively straightforward, however.

Current x86 implementations that use instruction compounding [2, 3, 4, 5] do so by fusing micro-ops from the same instruction or adjacent instructions in the decoded instruction stream. In the proposed future processor, this constraint is removed so that micro-ops from non-adjacent instructions can be fused. Then, higher quality micro-op level optimizations, including fusion, are implemented. Implementing these high quality optimizations can be a

[1]In this paper, the terms "compounding" and the more recent term "fusing" will be used interchangeably.

relatively complex task, however, so a key feature of the future processor is dynamic translation software that is concealed from all conventional software, including system software. In effect, the translation software becomes part of the processor design; i.e., collectively the hardware and software become a *co-designed virtual machine* (VM) [6, 7] implementing the x86 ISA.

Because software binary translation is used, a potential problem with the co-designed VM paradigm is high startup overhead caused by initial code translation. This overhead can be dramatically reduced by implementing a two level x86 ISA decoder. The first level decomposes and maps x86 instructions into RISC-like micro-ops in a more-or-less conventional manner. However, the micro-ops themselves belong to a vertical microcode-like instruction set (*V-code*) that is identical to the implementation ISA for the co-designed VM; that is, V-code becomes the target instruction set for software binary translation. In effect, the first level decoder is a hardware implementation of a binary translator to V-code, and it performs only a very minimum level of optimization. Then, the second level decoder maps the V-code into horizontal micro-ops (H-code) that actually control the execution pipeline. For fast software startup, the hardware x86-to-V-code decoder is employed. As program hotspots are found, however, VM software translates these frequently used code segments into fused, optimized V-code that is held in a code cache in main memory. When such pre-translated code sequences are subsequently encountered and fetched from the code cache, they bypass the first level decoder, leading to improved performance.

In this paper the architecture of a future processor is described in further detail. As discussed above, the overall implementation is based on three key features:

1. Instruction compounding with a collapsed 3-1 ALUs and pipelined two-cycle issue logic. This dynamic execution engine not only achieves higher performance, but also significantly reduces the pipeline backend complexity, e.g., the result forwarding network.

2. A co-designed VM for translating x86 instructions to V-code and implementing sophisticated fusing algorithms. The resulting compound instructions take full advantage of the microarchitecture features given above.

3. A two stage front-end decoder for fast startup. The first stage is a hardware binary translator from the x86 ISA to V-code; it does simple translation and little, if any, optimization. Program hotspots are translated by software into optimized V-code that bypasses the first level decoder when it is executed.

2. PROCESSOR OVERVIEW

The future processor has two major components in its co-designed VM implementation – the software dynamic binary translator/optimizer and the supporting hardware microarchitecture. The interface between the two is implementation instruction set; in the future processor the implementation instruction set is V-code as noted above.

The overall future processor implementation is illustrated in Figure 1. At the frontend of the processor is the two-level decoder [8]. The first-level decoder translates x86 instructions into the V-code ISA, and the second-level decoder generates the horizontal decoded control signals (H-code) used by the pipeline. A two-level decoder is well-suited to the complex x86 ISA. Compared with a single-level monolithic decode control table, the two-level decoder is smaller, and it makes the decoding logic more regular, flexible, and amenable to pipelining.

By employing a two-level decoder, both x86 code and V-code can be executed by the same pipeline. As the processor runs, it dynamically switches back and forth between x86 mode and V-code mode, under the control of co-designed VM software. When executing pre-translated V-code, the first level of decode is bypassed, and only the second (horizontal) decode level is used. When directly executing x86 code, instructions pass through both decode levels; for example, this would be done when a program starts up. In this mode, there is no dynamic optimization and minimal fusion, so performance will be similar to (or perhaps slightly less than) a conventional x86 implementation.

Profiling hardware such as that proposed by Merten et al. [9] detects frequently-used code regions (hotspots). As hotspots are discovered, the co-designed VM-software organizes them into superblocks [10], translates, and optimizes them as fused V-code. It then places the optimized V-code instructions into a code cache [11] held in a reserved area of main memory that is managed by concealed runtime software. Then, the full benefits of the fused V-code instruction set are realized. Note also that when optimized instructions are executed from the code cache, the first-level decode logic can be turned off for added power efficiency.

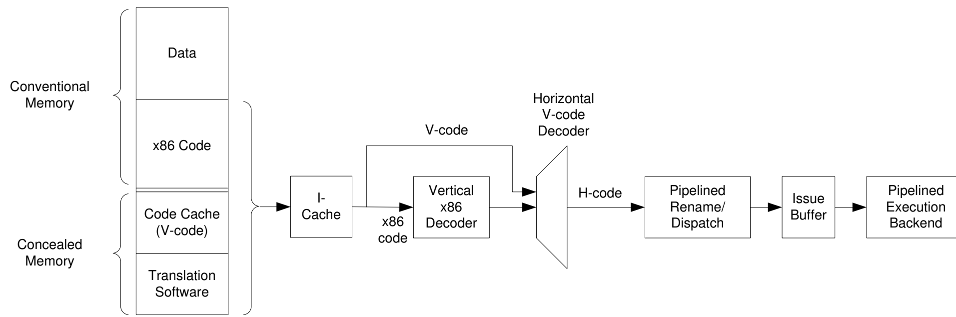

Fig. 1. Overview of proposed future superscalar processor implementation.

2.1. The Implementation ISA

A proposed implementation instruction set (V-code) is shown in Figure 2; it contains RISC-style micro-ops that target the x86 instruction set. As envisioned, the V-code ISA has both 16-bit and 32-bit formats. Using a mixed 16/32-bit format provides a denser encoding of translated instructions (and better I-cache performance) than a 32-bit-only format as in most RISCs (note that the classic RISC machines from CDC and Cray Research also used both short and long instruction formats). As envisioned, the 32-bit formats encode three register operands and/or an immediate value. The 16-bit formats use an x86-like 2-operand encoding in which one of the operands is both a source and a destination register.

The first bit of each V-code instruction, the F-bit, indicates whether it can be fused with the immediately following instruction to form a V-code *macro-op* composed of a pair of simple V-code instructions. The *head* of a fused macro-op is the first V-code instruction in the pair, and the *tail* is the second, dependent V-code instruction which consumes the value produced by the head. Because the F-bit is functionally a hint, the choice of which instructions to fuse becomes a joint hardware/software implementation decision. For example, to reduce pipeline complexity, e.g., in the rename and scheduling stages, one can restrict fusing only of dependent micro-op pairs that have a combined total of two or fewer unique input register operands. This assures that the fused macro-ops can be handled by instruction rename/issue logic of conventional complexity and an execution engine with a collapsed 3-1 ALU. In the implementation evaluated in this paper, such a restriction to two or fewer unique input operands is assumed.

Core 32-bit instruction formats

F	10b opcode	21-bit Immediate/Displacement		
F	10b opcode	16-bit immediate/Displacement		5b Rds
F	10b opcode	11b Immediate/Disp	5b Rsr	5b Rds
F	16-bit opcode	5b Rsr	5b Rsr	5b Rds

Add-on 16-bit instruction formats for code density

F	5b op	10b Immd/Disp	
F	5b op	5b Rsr	5b Rds
F	5b op	5b Rsr	5b Rds

Fig. 2. Proposed V-code Formats

2.2. Dynamic Binary Translator

The dynamic binary translator is implementation-dependent software, co-designed with the hardware implementation. In effect, it generates optimized V-code sequences from x86 machine code sequences. Although it can implement a number of optimizations [12, 11], the optimization we are most interested in here is micro-op compounding or fusing. Because of the presence of the dual decoder front-end, x86 instructions are initially executed

with minimal optimization being done by the hardware x86-to-V-code decoder. However, profiling hardware finds frequently executed code sequences and provides this information to the co-designed software.

Hotspot x86 instructions are first assembled as superblocks (straight-line code sequences with a single entry point and one or more exit points). Superblock formation, in itself, is a code optimization that improves spatial locality and reduces miss rate in the instruction cache. Within a given superblock, x86 instructions are first de-composed into the RISC-like V-code. Appropriate V-code pairs are then located, re-ordered, and fused into macro-ops (further detail is given in the next section). The straightened, translated, and optimized code for a superblock is placed in a concealed, non-architected area of main memory – the code cache.

Note that the native x86 CISC instruction set already contains what are effectively fused operations. However, the proposed dynamic binary optimizer often compounds V-code micro-op pairs in different combinations than the ones that appear in the original x86 code. The co-designed hardware and software also allow pairings of operation types that are not permitted by the native x86 instruction set; for example the pairing of two ALU operations and the fusing of condition test instructions with conditional branches. Finally, it is important to note that many other runtime optimizations can be performed by the dynamic translation software, e.g. performing common sub-expression elimination and implementing the Pentium M's "stack engine" [5] cost-effectively in software, or even conducting "SIMDification" [12] to exploit SIMD functional units.

2.3. Microarchitecture

The co-designed microarchitecture (Figure 1) has the same basic stages as a conventional x86 out-of-order superscalar pipeline. Consequently it inherits most of the proven benefits of such designs. The key difference is that the proposed microarchitecture can process instructions at the coarser granularity of fused macro-ops throughout the entire pipeline.

Because of the two-level decoders, the co-designed pipeline has two slightly different configurations – one when decoding x86 instructions in hardware and another when executing pre-decoded and optimized V-code from the code cache. For x86 code, the pipeline operates just as a conventional dynamic superscalar processor except that the instruction issue logic is pipelined for a faster clock cycle. Immediately following the initial decode stage, certain adjacent V-code micro-ops from x86 instructions can be re-fused as in some current x86 implementations, but no re-ordering or optimizations are done. The important point is that even when not running optimized V-code, the pipeline is still a high performance superscalar processor for x86 instructions.

For optimized V-code, paired dependent micro-ops are placed in adjacent memory locations in the code cache and are identified via the special F-bit. After they are fetched, the paired V-code instructions are immediately aligned together and fused into a single macro-op. Then macro-ops are processed throughout the rest of the pipeline as single units (Figure 3). By processing a fused micro-op pair as a unit, processor resources such as register ports and instruction dispatch/tracking logic are reduced and/or better utilized. Perhaps more importantly, the dependent V-code micro-ops in a fused pair share a single issue buffer slot and are awakened and selected for issue as a single entity. The number of issue window slots and issue width can then be reduced (below the number for separate micro-ops) without affecting performance.

As will be shown below, after fusing, there are very few macro-ops that contain an isolated single-cycle ALU operation. Consequently, key pipeline stages can be designed as if the minimum instruction execution latency is two cycles. The instruction issue stage, for example, can be simplified because of the significantly reduced importance of issuing single cycle back-to-back instructions. This means that the instruction issue stage can be pipelined in two stages, simply and with minimal performance loss. Another critical pipeline stage is the ALU. In the proposed future processor, two dependent ALU micro-ops in a macro-op can be executed in a single cycle by using a combination of a collapsed three-input ALU [13, 14, 15] and a conventional two-input ALU. If most single-cycle ALU operations are fused with other operations, then the expensive ALU-to-ALU operand forwarding network can be eliminated. What is ordinarily the ALU stage can take a total of two cycles, with the actual ALU operation(s) in the first cycle, and most, if not all, of the second cycle being used for writing results back to the registers where they can be accessed by dependent operations. Besides pipeline simplification, there are other ways to simplify a CISC microarchitecture in a co-designed VM implementation. For example, unused legacy features in the architected ISA can be largely (or entirely) emulated by translation software. A simple microarchitecture reduces design risks and cost, and yields a shorter time-to-market. Although the translation software must be validated for correctness, this software does not require physical design checking, does not require circuit timing verification, and if a bug is discovered late in the design process, it does not require re-spinning the silicon.

usage. After building the dependence graph, the two-pass fusing algorithm looks for pairs of dependent single-cycle ALU micro-ops during the first scan. In the example, the AND micro-op and the first ADD micro-op are fused. (Fused pairs are marked with double colon, :: in Figure 5c). Re-ordering, as is done here, complicates precise traps because the AND micro-op overwrites the value in register EAX earlier than in the original code. Register assignment that extends live ranges resolves this issue [18]; i.e., R20 is assigned to hold the result of the first ADD, retaining the original value of EAX. During the second scan, the fusing algorithm considers multi-cycle micro-ops (e.g., memory ops) as candidate tails. In this pass, the last two dependent micro-ops are fused as an ALU-head, LD-tail macro-op. The key for fusing macro-ops is to fuse more dependent pairs on or near the critical path. A

```
1. lea     eax, DS:[edi + 01]
2. mov     [DS:080b8658], eax
3. movzx   ebx, SS:[ebp + ecx ≪ 1]
4. and     eax, 0000007f
5. mov     edx, DS:[eax + esi ≪ 0 + 0x7c]
```

(a) x86 assembly

```
1. ADD     Reax, Redi, 1
2. ST      Reax, mem[R18]
3. LDzx    Rebx, mem[Rebp + Recx ≪ 1]
4. AND     Reax, 0000007f
5. ADD     R21, Reax, Resi
6. LD      Redx, mem[R21 + 0x7c]
```

(b) micro-operations

```
1. ADD     R20, Redi, 1                  ::   AND Reax, R20, 007f
2. ST      R20, mem[R18]
3. LDzx    Rebx, mem[Rebp + Recx ≪ 1]
4. ADD     R21, Reax,Resi                ::   LD Redx, mem[R21 + 0x7c]
```

(c) Fused macro-ops

Fig. 5. Two-pass fusing algorithm example.

two-pass fusing algorithm fuses a high percentage of single-cycle ALU pairs on the critical path by observing that the criticality for ALU-ops is easier to model and that fused ALU-ops better match the collapsed ALU units.

3.2. Evaluation

The fusing algorithm was evaluated using the SPEC2000 Integer benchmarks. Results are given in Figure 6. The degree of fusing, i.e., the percentage of micro-ops that are fused into pairs determines how effectively the macro-op mode can utilize the pipeline bandwidth. Furthermore, the profile of non-fused operations implies how the pipelined issue logic may affect IPC performance. Figure 6 shows that on average, more than 56% of all dynamic micro-ops are fused into macro-ops. Most of the non-fused operations are loads, stores, branches, floating point, and NOPs. Non-fused single-cycle integer ALU micro-ops are only 6% of the total, thus greatly reducing the possible penalty due to pipelining the issue logic. The nearly 60% fused micro-op pairs lead to an effective 30% bandwidth reduction through the pipeline.

Additional fusing characterization data (note shown here) were also collected on the SPEC2000 integer benchmarks to evaluate the fusing algorithm and its implications on the co-designed pipeline. Nearly 70% of the fused macro-ops are composed of micro-ops from two different original x86 instructions, suggesting that inter-x86 instruction optimization is important at runtime. Among the fused macro-ops, more than 50% are composed of two single-cycle ALU micro-ops, about 18% are composed of an ALU operation head and a memory operation tail, about

	BASELINE	MACRO-OP
ROB Size	128	128
Retire width	3,4	2,3,4 MOP
Issue Pipeline Stages	1	2
Fuse RISCops?	No	Yes
Issue Width	3,4	2,3,4 MOP
Issue Window Size	Variable. Sample points: from 16, up to 64. Effectively larger for the macro-op mode.	
Register File	128 entries, 8,10 Read ports, 5,6 Write ports	128 entries, 6,8,10 Read and 6,8,10 Write ports
Functional Units	4,6,8 INT ALU, 2 MEM R/W ports, 2 FP ALU	
Cache Hierarchy	4-way 32KB L1-I, 4-way 32KB L1-D, 8-way 1 MB L2	
Cache/Memory Latency	L1 : 2 cycles + 1 cycle AGU, L2 : 8 cycles, Mem: 200 cycles for the 1st chunk, 6 cycles b/w chunks	
Fetch width	16-Bytes x86 instructions	16-Bytes V-code

Table 1. Microarchitecture Configurations

Steady State Performance

Figure 7 shows the relative IPC performance for issue window sizes ranging from 16 to 64. Performance is normalized with respect to a 4-wide baseline x86 processor with a size 32 issue window[2]. Five bars are presented for configurations of 2-, 3-, and 4-wide *macro-op* execution model; 3- and 4-wide *baseline* superscalar. If we first focus on

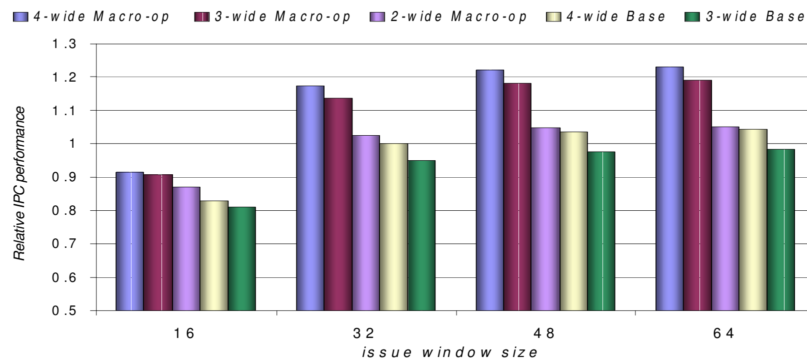

Fig. 7. IPC performance comparison

complexity effectiveness, we observe that the two-wide co-designed x86 implementation performs at approximately the same IPC level as the four-wide baseline processor. However, the two-wide macro-op model has approximately same level of complexity as a conventional two-wide machine. The only exceptions are stages where individual micro-ops require independent parallel processing elements, i.e. ALUs. Furthermore, the co-designed x86 processor pipelines the issue stage by processing macro-ops. Hence, the macro-op model should be able to support either a significantly higher clock frequency or a larger issue window for a fixed frequency, while giving the same or better IPC performance as a conventional four-wide processor. It assumes a pipeline no deeper than the baseline model, and in fact it reduces pipeline depth for hotspot code by removing the complex first-level x86 decoding stage(s) from the critical branch misprediction redirect path. For reference, if the issue logic is pipelined for a faster clock in the baseline design, there is an IPC performance loss about $6 \sim 9\%$.

[2]The normalized values are very close to the absolute values; the harmonic mean of absolute x86 IPC is 0.96 for the four-wide baseline with issue window size 32.

timizers, this approach achieves high performance when in optimized macro-op mode with minimal performance loss in during startup. This is especially important for the x86 (and CISC ISAs in general) which generates many micro-op sequences that are not optimized. In addition, the co-designed paradigm opens up further dynamic optimization opportunities that remain to be explored. An important feature of the future processor is the dual decoder frontend, which dramatically reduces the startup overhead normally experienced with a software-only co-designed VM.

The future processor design improves processor efficiency by reducing pipeline stage complexity for a given level of IPC performance. For complexity effective processor designs, a two-wide future processor significantly reduces pipeline complexity without losing IPC performance when compared with a four-wide conventional super-scalar pipeline. The biggest complexity savings are in the form of reduced pipeline width, pipelined instruction issue logic, and the removal of ALU-to-ALU forwarding paths. This reduced complexity will lead to a higher frequency clock (and higher performance), reduced power consumption, and shorter hardware design times.

Alternatively, with similar design complexity to a conventional processor, the future processor's macro-op execution engine improves IPC performance by an average of 20% over a comparable conventional superscalar design on integer benchmarks. From the IPC perspective, the largest performance gains come from operation compounding which treats fused micro-op pairs as single entities throughout the pipeline to improve ILP and reduce communication and management overhead. Simulation data shows that there is a high degree of macro-op fusing in typical x86 binaries, and this improves throughput for a given macro-op pipeline width and scheduling window size.

6. ACKNOWLEDGEMENTS

At the heart of the future superscalar processor are a number of concepts pioneered by Stamatis Vassiliadis. Other researchers who have contributed to the proposed future superscalar processor include Shiliang Hu, Mikko Lipasti, Ho-Seop Kim, Ilhyun Kim, and Yanos Sazeides.

7. REFERENCES

[1] S. Vassiliadis, B. Blaner, and R. J. Eickemeyer, "SCISM: A Scalable Compound Instruction Set Machine," *IBM Journal of Research and Development*, pp. 59–78, January 1994.

[2] K. Diefendorff, "K7 Challenges Intel," *Microprocessor Report*, vol. Vol.12, No. 14, Oct. 25, 1998.

[3] C. N. Keltcher et al., "The AMD Opteron Processor for Multiprocessor Servers," in *IEEE MICRO*, Mar.-Apr. 2003, pp. 66 –76.

[4] I. Kim and M. H. Lipasti, "Macro-op Scheduling: Relaxing Scheduling Loop Constraints," in *Proc. of the 36th Int'l Symp. on Microarchitecture*, Dec. 2003, pp. 277–288.

[5] S. Simcha Gochamn et al., "The Intel Pentium M Processor: Microarchitecture and Performance," *Intel Technology Journal*, vol. Vol 7, Issue 2, 2003.

[6] K. Ebcioglu and E. R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," in *Proc. of the 24th Int'l Symp. on Computer Architecture*, 1997.

[7] A. Klaiber, "The Technology Behind Crusoe Processors," *Transmeta Technical Brief*, 2000.

[8] E. P. Stritter and H. L. Tredennick, "Microprogrammed Implementation of a Single Chip Microprocessor," in *Proc. 11th Annual Microprogramming Workshop*, Nov. 1978, pp. 8–16.

[9] M. Merten et al., "An Architectural Framework for Runtime Optimization," *IEEE Trans. Computers*, vol. 50(6), pp. 567–589, 2001.

[10] W.-M. Hwu, S. A. Mahlke, and W. Y. Chen, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *The Journal of Supercomputing*, vol. 7(1-2), pp. 229–248, 1993.

[11] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A Transparent Dynamic Optimization System," in *Int'l Symp. on Programming Language Design and Implementation*, Jun. 2000, pp. 1–12.

[12] Y. Almog et al., "Specialized Dynamic Optimizations for High-Performance Energy-Efficient Microarchiteture," in *Proc. of the 2nd Int'l Symp. on Code Generation and Optimization*, 2004.

[13] N. Malik, R. J. Elckemeyer, and S. Vassilladis, "Interlock Collapsing ALU for Increased Instruction-Level Parallelism," *ACM SIGMICRO Newsletter*, vol. Vol. 23, pp. 149 – 157, Dec. 1992.