The MAJC Architecture: A Synthesis of Parallelism and Scalability

THE MAJC ARCHITECTURE ENHANCES APPLICATION PERFORMANCE BY EXPLOITING PARALLELISM AT MULTIPLE LEVELS—INSTRUCTION, DATA, THREAD, AND PROCESS. SUPPORTING VERTICAL MULTITHREADING, SPECULATIVE MULTITHREADING, AND CHIP MULTIPROCESSORS, THE SCALABLE VLIW ARCHITECTURE IS ALSO CAPABLE OF ADVANCED SPECULATION AND PREDICATION AND TREATS ALL DATA TYPES SIMILARLY.

Marc Tremblay
Jeffrey Chan
Shailender Chaudhry
Andrew W. Conigliaro
Shing Sheung Tse
Sun Microsystems

• • • • • • The advent of broadband services, including HDTV-quality video on demand, voice navigation, virtual worlds, e-distribution of new media content, and 3D browsing, has had a profound impact on microprocessor workloads. In the broadband-service world, a transaction no longer consists of a trivial record update or simple arithmetic on a bank account. Most likely, the transaction must be decrypted and decompressed. It may require signal processing—for instance, if it's a voice request. The transaction may have to meet real-time requirements to provide a certain quality of service, and it may require a reverse sequence of operations to return the result to a user or another computer. Such transactions are fundamentally different from the workloads and dynamic instruction profiles—such as those extracted from the Sieve and Tower of Hanoi benchmarks—used to develop the CISC and RISC architectures of the 1970s and 1980s.

With these extreme computing requirements in mind, Sun Microsystems developed the Microprocessor Architecture for Java Computing (MAJC, pronounced "magic") to meet the needs of general-purpose computing in the coming years. First, our team of architects emphasized broadband-service processing requirements. Second, because reports indicated that many applications handled by application servers and Web servers are written in Java, we wanted to enhance Java performance. Current microprocessor architectures tend to limit the performance level of these applications. Typically, the limitations stem primarily from a lack of thread-level parallelism, a lack of support for a strict exception model, and deficient microprocessing scalability. MAJC addresses these issues through its innovative design.

We also wanted to improve the performance of applications and system software written in languages such as C and C++. So we designed MAJC to exploit scalable parallelism—not only at the instruction level, but the data, thread, and process levels as well. Finally, we wanted the ability to develop MAJC implementations quickly and efficiently within the limits of a given power budget and a high clock rate. Thus, we designed

0272-1732/00/$10.00 © 2000 IEEE

a flexible architecture that provides compatibility between implementations through simple binary translation and compatibility with existing instruction set architectures through dynamic binary translation[1,2] (much like the dynamic translation of Java bytecodes to the MAJC ISA through Sun's HotSpot virtual machine[3]). This article describes how we met these goals through MAJC's architectural features, instruction set, and thread-level support.

Architectural features

Given the opportunity to define a new microprocessor architecture from scratch, we reexamined the assumptions shaping current architectures and analyzed the latest research results. After choosing the best features from the current crop of microprocessors, we developed features beyond the state of the art to incorporate in MAJC. To achieve a streamlined architecture, we included only features that would facilitate our goals.

MAJC is a scalable, very-long-instruction-word (VLIW) architecture[4] with a hierarchical structure. The lowest level consists of instruction slices that provide the control, status, and architectural registers and the resources required to execute instructions in a VLIW packet. One to four instruction slices, as well as additional control and status registers including the program counter register and the processor unit status register, combine to form a microthread for executing a single VLIW packet stream. One or more microthreads and additional control and status registers, such as the processor unit control register, combine to form a processor unit. Finally, one or more processor units and additional control and status registers combine to form a processor cluster. The architecture supports fast communication between microthreads and processor units in a processor cluster through fast interrupts and virtual channels.

This structure supports parallelism at several levels:

- data-level parallelism through single-instruction, multiple-data (SIMD) instructions,[5]
- instruction-level parallelism through VLIW packets containing from one to four instructions, and
- thread-level and process-level parallelism through vertical multithreading[6,7] and fast communication between chip multiprocessors (CMPs).[8]

Dependency checking and resource allocation move from hardware to software, simplifying MAJC implementations and improving time to market.

Even with recent research advances in instruction-level parallelism, high-frequency execution remains a dominant factor determining application performance in modern general-purpose microprocessors. Although issuing multiple instructions in the same cycle is an important performance-enhancing technique for most traditional architectures, it typically requires complex logic to handle data and control dependencies and resource allocation. The complexity of this logic has a significant impact on frequencies and on the time required to implement and verify the logic.

MAJC supports issuing multiple instructions in a single cycle with the VLIW approach, in which multiple instructions execute as a single packet. The instructions in each VLIW packet must have no data or control interdependencies and no resource-use interdependencies. A compiler for a MAJC microprocessor is responsible for ensuring that VLIW packets meet these rules.

The compiler must also perform inter-packet dependency checking and resource allocation, except in the cases of variable-latency instructions such as memory instructions, and complicated, extended-latency instructions such as division instructions. Thus, dependency checking and resource allocation move from hardware to software, simplifying MAJC implementations and improving time to market (generally, a one-month time-to-market delay requires a compensatory performance increase of approximately 4%). To

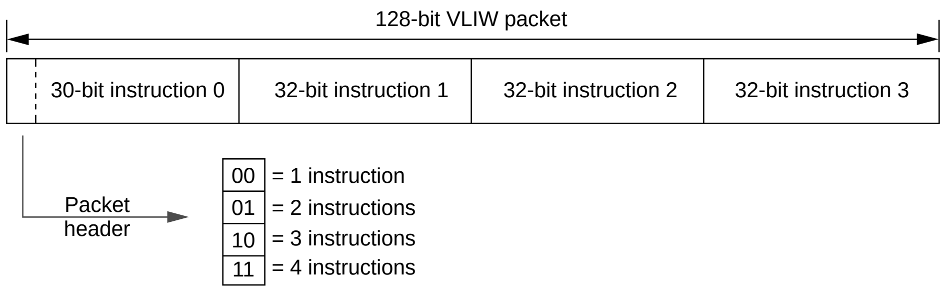


Figure 1. A VLIW packet as a variable-length-instruction-word packet.

schedule code efficiently for most modern microprocessors, compilers must have intimate knowledge of the microprocessors' pipelines, so dependency checking and resource allocation do not significantly increase the MAJC compiler's complexity.

Each MAJC VLIW packet contains from one to four 32-bit instructions. Traditional VLIW architectures require each VLIW packet to contain the maximum number of instructions even if there is insufficient instruction-level parallelism to actually issue that number of instructions per cycle. The unused instructions must be padded with no-operation instructions, which increase the executable's size, significantly affecting cache performance and memory bandwidth. In contrast, MAJC uses an encoding in each VLIW packet to indicate how many instructions the packet contains, thus reducing the executable footprint. Therefore, in MAJC, the term *VLIW packet* can also denote a variable-length-instruction-word packet.

A MAJC executable's size is very close to that of a RISC executable given support for similar 32-bit instructions. For example, the main loop in the SPEC CINT95 benchmark 129.compress contains 56 MAJC VLIW packets. A traditional four-issue VLIW architecture (chosen for comparison because the maximum MAJC packet size is four instructions) requires 224 instruction words in the loop. However, with the variable-packet-size approach, MAJC requires only 62 instruction words in the loop, a reduction of more than 70% of the memory footprint, thereby reducing cache and memory bandwidth requirements.

Conceptually, each instruction in a VLIW packet uses an instruction slice for its execution. Except for the first slice, the resources in each instruction slice are similar to each other. The first instruction in a VLIW packet has a different instruction slice because it has only one quarter of the opcode space available to other instructions (see Figure 1). Part of its opcode space is the VLIW packet header, which encodes the number of instructions available in the VLIW packet.

In many ways, each instruction slice is an instruction's independent execution path. Each instruction slice has its own control and status registers, as well as private registers. Instructions execute completely within their own slices without requiring the resources in another slice, reducing communication requirements between instruction slices and, thus, the wires between slices. However, the global registers are shared so that instruction slices can communicate with each other. An implementation designer can choose to replicate these registers per slice and to broadcast updates from the other slices to maintain each slice's independence.

Instruction slices enable an implementer to use copy-and-paste techniques to design similar instruction slices, reducing design and verification efforts. The architecture lets the implementer choose whether an implementation supports a maximum of one, two, three, or four instructions in a VLIW packet to address a particular application requirement. The copy-and-paste technique supports this freedom by allowing additional instruction slices without significant additional effort. Binary translation is the planned method for maintaining binary compatibility between implementations.

MAJC strikes a balance between efficiently processing traditional applications, without leaving many instruction slices idle, and speeding up the processing of broadband applications. (By traditional applications, we mean SPEC CINT-type applications, which

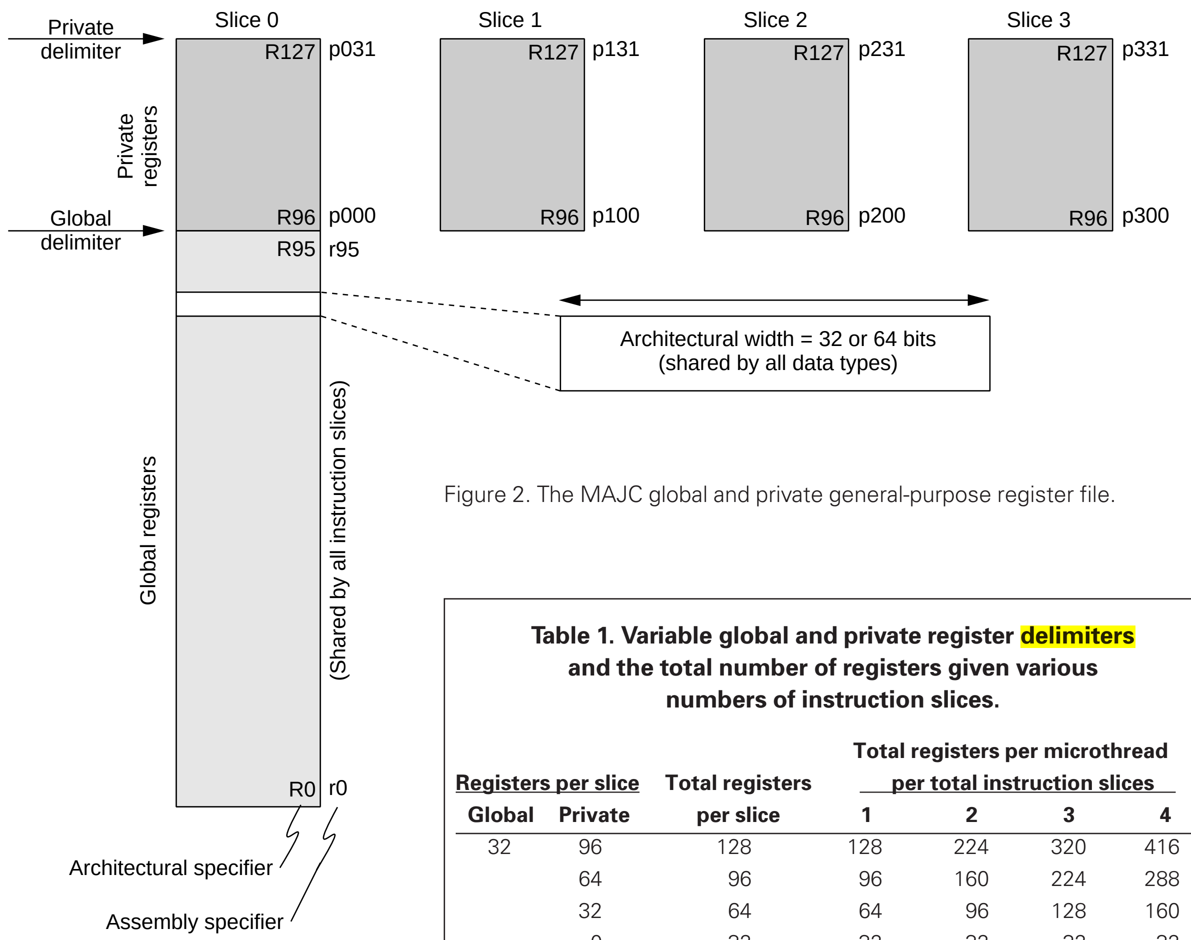


Figure 2. The MAJC global and private general-purpose register file.

Table 1. Variable global and private register delimiters and the total number of registers given various numbers of instruction slices.

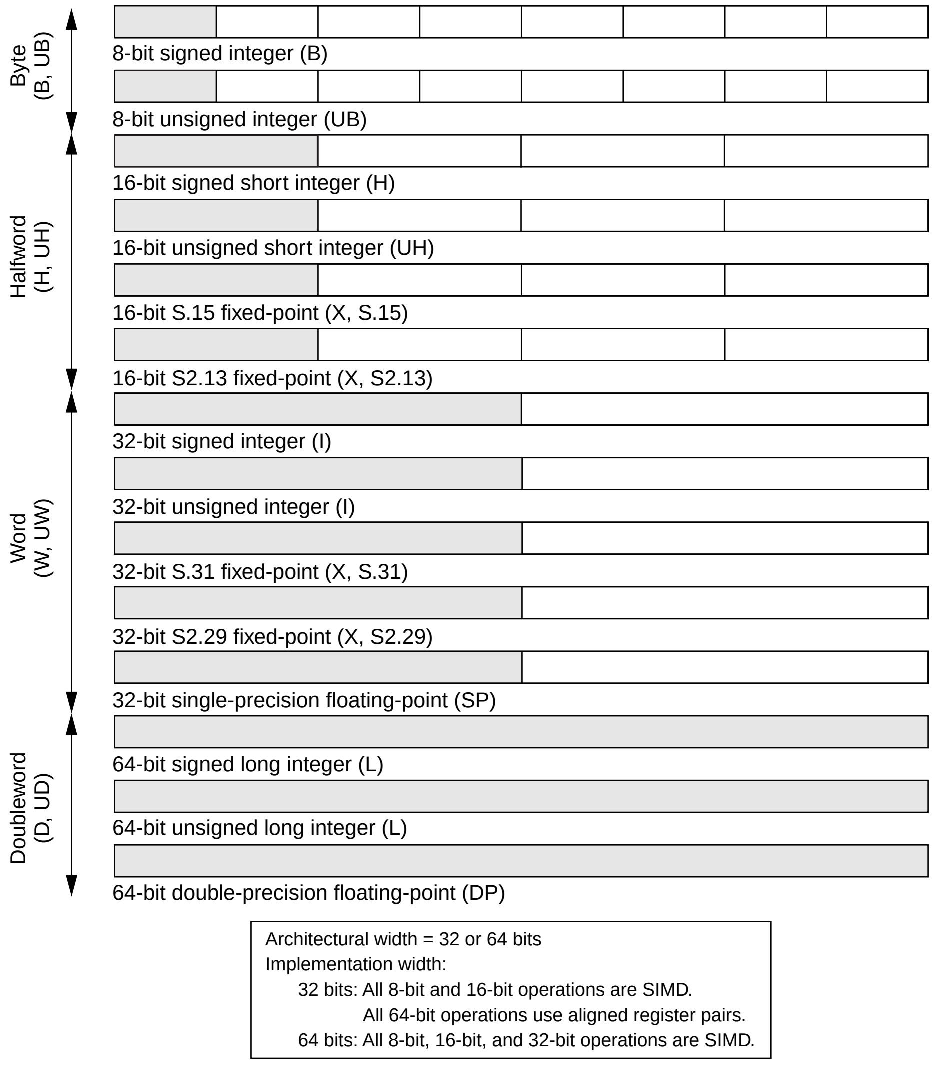
Registers per slice		Total registers	Total registers per microthread per total instruction slices			
Global	Private	per slice	1	2	3	4
32	96	128	128	224	320	416
	64	96	96	160	224	288
	32	64	64	96	128	160
	0	32	32	32	32	32
64	64	128	128	192	256	320
	32	96	96	128	160	192
	0	64	64	64	64	64
96	32	128	128	160	192	224
	0	96	96	96	96	96
128	0	128	128	128	128	128

are a representative sample of general-purpose computing applications.) The architecture supports a maximum of four 32-bit instructions in a VLIW packet because extracting instruction-level parallelism beyond four instructions in traditional applications is relatively difficult. However, each instruction can be SIMD, allowing up to four different SIMD instructions in a VLIW packet. This multiple-SIMD capability, coupled with the uniformity of resources in each instruction slice, provides application programmers the flexibility to fully use all the instruction slices. Thus, MAJC significantly speeds up broadband applications without supporting more instructions in a VLIW packet.

MAJC specifies that each instruction can access up to 128 registers, requiring 7 bits per register specifier to encode. An implementation can allow access to 32, 64, 96, or 128 registers per instruction (see Figure 2 and Table 1). These registers consist of global and private registers. Global registers are shared by all the instruction slices to permit interslice communication. Private registers are accessible only by instructions in the same slice.

The number of global and private registers per instruction slice is implementation-dependent as long as the implementation meets the following rules:

- there is a minimum of 32 global registers,
- the number of global registers is a multiple of 32,
- the number of private registers is a multiple of 32, and


Byte (B, UB)
8-bit signed integer (B)
8-bit unsigned integer (UB)
Halfword (H, UH)
16-bit signed short integer (H)
16-bit unsigned short integer (UH)
16-bit S.15 fixed-point (X, S.15)
16-bit S2.13 fixed-point (X, S2.13)
Word (W, UW)
32-bit signed integer (I)
32-bit unsigned integer (I)
32-bit S.31 fixed-point (X, S.31)
32-bit S2.29 fixed-point (X, S2.29)
32-bit single-precision floating-point (SP)
Doubleword (D, UD)
64-bit signed long integer (L)
64-bit unsigned long integer (L)
64-bit double-precision floating-point (DP)

Architectural width = 32 or 64 bits
Implementation width:
32 bits: All 8-bit and 16-bit operations are SIMD.
All 64-bit operations use aligned register pairs.
64 bits: All 8-bit, 16-bit, and 32-bit operations are SIMD.

Figure 3. MAJC data types.

- the sum of the number of global and private registers per instruction slice does not exceed 128.

Thus, with a maximum of four instruction slices per VLIW packet, the MAJC register file definition allows from 32 to 416 registers but requires only 7 bits of opcode space per register specifier. For example, a VLIW packet supporting three instruction slices with 64 private registers per slice and 32 global registers has a total of 224 registers. Likewise, a VLIW packet supporting four instruction slices with 32 private registers per slice and 96 global registers also has a total of 224 registers.

The private registers accessed by an instruction depend on the instruction slice that the instruction in the VLIW packet occupies. MAJC's register configuration flexibility permits implementation trade-offs between the area and cycle-time impact of using a large number of registers and the performance achieved by using a register configuration designed for a particular application.

The MAJC architectural registers as well as the instruction slices are "data-type agnostic"; that is, they can store and process information regardless of its data type. The instruction set provides a rich set of instructions to process a variety of data types ranging from 8- to 64-bit data types (see Figure 3). An implementation can choose either 32- or 64-bit register widths. MAJC treats data types smaller than an implementation's register width as SIMD data types. Data types larger than an implementation's register width require a pair of evenly aligned registers to hold the data.

Unlike other architectures, which typically have separate integer, floating-point, and SIMD registers and execution resources, MAJC integrates the processing of various data types in an instruction slice. The compiler can then allocate registers for program variables without regard to data type and can efficiently allocate resources to meet an application's requirements. In addition, because a VLIW packet need not dispatch instructions to separate execution resources based on the data type it is processing, instruction processing in an instruction slice is streamlined. An instruction slice can share the execution resources for processing various data types without replicating common resources such as multiplier arrays, leading to better resource use.

The architecture specifies a precise execution model in which each VLIW packet executes to completion or does not execute at all in sequential program order. All the instructions in a VLIW packet execute at the same time, so they must not have any read-after-write or write-after-write interdependencies. Write-after-read dependencies between instructions in the same VLIW packet are permitted. Instructions in a VLIW packet generate precise exceptions before the packet updates an architectural state. Interrupts occur asynchronously after a VLIW packet has completed updating any architectural state, but before the next sequential program order VLIW packet updates any architectural state.

MAJC supports three trap levels, thus allowing two nested traps. The privilege mode of software executing in a trap level is controlled by a field in the corresponding trap level's processor unit status register and is independent of the trap level. User programs and system software will execute at the first trap level—the former in nonprivileged mode and the latter in privileged mode. The second trap level is for interrupt and exception dispatch software, and the third trap level is for diagnostic software.

The memory consistency model assumed by MAJC is a variant of release consistency.[9] In this model, the sole constraint is that memory operations on the same address from a single execution stream appear to be performed in sequential program order with respect to that stream. However, VLIW packet fetching is not coherent with the memory operations of an execution stream. Memory operations on different addresses from an execution stream are independent and thus appear to be unordered. Additionally, there is no total order between memory operations from different execution streams. To order memory operations, MAJC requires the use of appropriate memory barrier instructions to provide the semantics of acquire and release operations as defined by release consistency. Release consistency provides the greatest flexibility for implementers to design memory subsystems for CMP and multiprocessor systems efficiently. Since Java programs already assume a relaxed memory model,[10] the architecture does not additionally burden application programmers by using a relaxed memory model such as release consistency.

Instruction set

The MAJC instruction set supports a load-store architecture, which loads information from memory into general-purpose registers before processing it and stores information back to memory from registers after processing it. The instruction set provides a rich set of operations to process information in the MAJC register file. Because the general-purpose registers are data-type agnostic, the instruction set must provide several variations of many instructions to process various data types. The instruction slices too are data-type agnostic. They allow an instruction to execute in any instruction slice, with the restriction that the first instruction in a VLIW packet has only one quarter of the instruction opcode space. Thus, not all operations are available to instructions for the first slice.

The instruction set provides instructions for typical RISC operations such as add and shift, as well as operations that improve the performance of both broadband-service transaction processing and Java programs. It also supports instructions for predicated and speculative execution so that the compiler can extract greater instruction-level parallelism from traditional programs.

Often, traditional instruction set architectures cannot efficiently handle modern applications. One way that MAJC counters this inefficiency is by providing an ISA capable of handling instructions with four register specifiers. With four register specifiers, the instruction set can include instructions especially important for multimedia applications, such as muladd (fused multiplication and addition), bitext (bit extract), byteshuffle, and pickc (conditional pick). These operations, which require three source operands and one destination operand, could not be implemented in traditional architectures.

Along with greater flexibility for creating a diverse and powerful instruction set, MAJC potentially provides many more general-purpose registers than a typical architecture. As previously described, up to 416 data-type-agnostic general-purpose registers let complex algorithms execute completely from the register file. This eliminates the need for additional memory instructions to fill and spill variables into and out of the register file, as would be necessary in architectures that do

A MAJC implementation example

Figure A shows a MAJC implementation example. In this configuration, the processor cluster consists of four processor units. PUs 0 and 1 are four-issue units, each containing, say, 96 global general-purpose registers and 32 private general-purpose registers per instruction slice. PUs 2 and 3 are two-issue units, each containing, say, 32 global general-purpose registers and no private general-purpose registers. Therefore, this processor cluster contains 512 general-purpose registers.

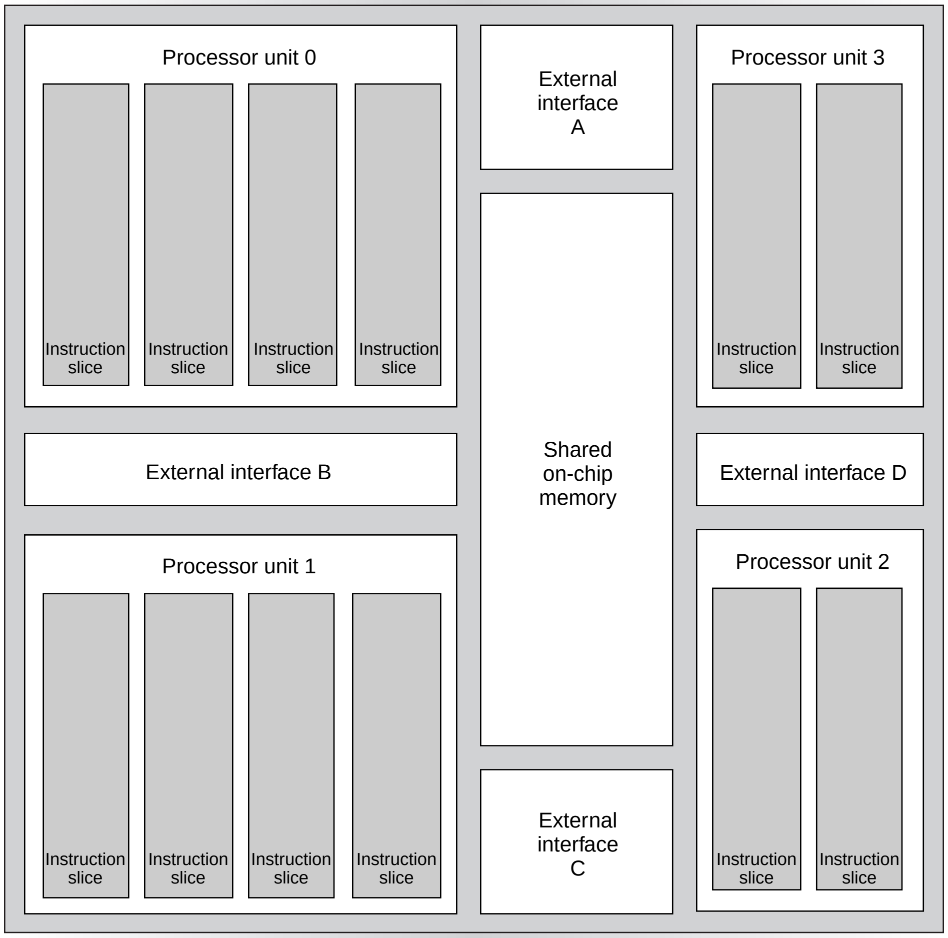


Figure A. An example of a four-processor-unit MAJC implementation.

not support many registers. Thus, the expansive MAJC register file results in higher application performance.

In accordance with our goal of exploiting instruction-level parallelism, the MAJC instruction set provides instructions to aid the compiler's predication and speculation techniques. The pickc (conditional pick), movc (conditional move), and stcd (conditional store doubleword) instructions, for instance, support speculatively executing code, thus extracting greater instruction-level parallelism. To fully utilize the instruction slices in an implementation, the compiler can hoist loads by executing them speculatively with nonfaulting load instructions.

Beyond instruction-level parallelism, the MAJC instruction set also includes instructions to enhance the performance of Java applications. The blkzero (block zero) instruction writes zero to a region of memory. Typically, Java virtual machines initialize large memory blocks. To accomplish this, most microprocessors use stores, which may unnecessarily transfer data from main memory into a cache. Blkzero optimizes this process by removing the unnecessary memory accesses. Another instruction that enhances Java application performance is bndchk (bound check). This instruction checks three conditions necessary for an array access in Java and causes a trap if any of them is true. This single instruction achieves an operation that requires numerous instructions, including calculations, comparisons, and branches, in most other instruction sets.

Moreover, MAJC includes stiw (store instruction word). This instruction is targeted at the self-modifying code feature of many Java virtual machines, which replaces certain instructions with other instructions during runtime for performance enhancement. Unlike normal store instructions, stiw snoops the VLIW packet stream. Without stiw, normal stores would be forced to snoop the VLIW packet stream to appropriately handle self-modifying code. Stiw makes this overhead unnecessary.

Beyond Java-enhancing instructions, the MAJC instruction set provides a set of powerful instructions for broadband-service transaction processing. These instructions include various single- and double-precision floating-point arithmetic operations such as maximum, minimum, absolute value, negate value, and reciprocal square root. These operations form a math library that facilitates graphics and multimedia applications. Individually, the maximum and minimum operations are very useful for voice recognition applications, and the reciprocal square root operation is very useful for vector normalization in graphics. MAJC also provides four-operand muladd and mulsub instructions. The floating-point versions of these instructions support graphics transformations, and the integer versions support graphics lighting computations. Another powerful instruction is the floating-point clip operation, which determines the relationship between two single-precision floating-point numbers to support graphics window clipping.

For 8-bit, 16-bit, and, if the architectural width is 64, 32-bit operations that would not fully utilize the data path on their own, the MAJC instruction set provides SIMD instructions important in media processing. These include addition, subtraction, mean value, power, shift, compare, move, fused multiplication and addition, dot-product-with-addition, and dot-product-with-subtraction calculations. The powerp (parallel power) instruction works well in graphics lighting, and the dotaddp and dotsubp instructions support 16-bit integer fast Fourier transforms and inverse discrete cosine transforms for MPEG-2 applications. The meanp instruction performs motion compensation calculations in MPEG-2 and DVD applications.

In addition to SIMD instructions, byte- and bit-manipulation instructions strengthen the instruction set. The cccb instruction, for instance, which counts consecutive clear bits, is useful in algorithms manipulating entropy code data and in variable-length decoding in MPEG-2 and DVD applications. The bitext instruction extracts bits from given data, another operation that is useful in variable-length decoding. The byteshuffle and pack instructions shuffle bytes of data and pack data, respectively, for motion compensation in MPEG-2 applications.

Table 2 (next page) shows the MAJC instruction set. Because MAJC instruction slices can operate on multiple data types, the table doesn't list instructions for different data types under separate headings. For example, instructions with the ADD prefix perform addition for halfwords, integers, long integers, and single- and double-precision floating-point numbers on the same instruction slices. Moreover, variations of instructions with the ADD prefix perform parallel operations (SIMD), use various rounding methods, and execute with saturation. There are combinations of these variations such as addhs, which adds two 16-bit halfwords in parallel with saturation.

Through the instruction set enhancements, multimedia and graphics applications can achieve significantly better performance on a MAJC microprocessor than on traditional general-purpose microprocessors.

Support for threads

The popularity of Java, which provides native support for multithreaded programs, has increased the availability of such programs. The use of multiprocessor systems is growing rapidly to meet the Internet's increasing data-processing demands. Broadband services require even greater processing capability. Typically, these services are multithreaded or contain multiple processes. To address these types of applications, MAJC supports thread- and process-level parallelism.

As previously mentioned, MAJC supports processor units containing multiple microthreads. This scheme supports vertical multithreading by having multiple execution streams share the resources in a processor unit on a cycle-by-cycle basis. Vertical multithreading allows more efficient use of a processor unit's resources by allowing another thread to use resources that a stalled thread is not using. In turn, processing throughput increases when there are multiple threads or processes to execute. With an increasing number of transactions on the Internet, each fairly independent of other transactions and capable of being processed with separate threads, throughput becomes increasingly important.

Additionally, MAJC has processor clusters that contain multiple processor units. These processor clusters support CMPs, which have drawn much attention from microprocessor architects in industry and academia. As process technology improves as predicted by Moore's law, CMP technology is one way to use the huge number of transistors available

Table 2. The MAJC instruction set.

Prefix	Size(s)*	No. of instructions	Description	Variations**
ABS	SP, DP	2	Absolute value	par
ADD	H, I, L, SP, DP	14	Add	par, rnd, sat
AND	W	1	And	
B		6	Branch	cond1, pred
BITEXT	D	1	Bit extract	
BITINSRT	W	1	Bit insert	
BLKZERO		1	Block zero memory	
BNDCHK		1	Bound check	
BYTESHUFFLE	D	1	Byte shuffle	
CALL		1	Call	
CAS	W	1	Atomic compare-and-swap	
CCCB		1	Count consecutive clear bits	
CLIP	SP	1	Clip value	par
CMP	B, I, L, SP, DP	24	Compare	cond2, par
CVT	I, L, SP, DP, X	28	Convert	par, rnd
DIV	I, L, SP, DP	12	Divide	rnd, s/us
DOTADD	H/I	1	Parallel dot product with addition	
DOTSUB	H/I	1	Parallel dot product with subtraction	
FLUSHI		1	Flush instructions	
GETIR		1	Get internal register	
JMPL		3	Jump-and-link	cond4
LD	B, H, W, D, G	32	Load register(s)	alt, NF, s/us
MAX	B, SP, DP	3	Maximum	par
MEAN	UH	1	Parallel arithmetic mean	par
MEMBAR		1	Memory barrier	
MIN	B, SP, DP	3	Minimum	par
MOVC	B, H, W	6	Move conditional	par, cond1
MUL	H, I, SP, DP	15	Multiply	par, cond3, rnd, sat
MULADD	I, SP, DP	5	Multiply and add	par, sat
MULSUB	I, SP	2	Multiply and subtract	par
NEG	SP, DP	2	Negate	
NOP		1	No operation	

(continued on next page)

in next-generation microprocessors to process more data faster. In other words, it is a method of increasing throughput.

From a software perspective, vertical multithreading and CMPs are ideal platforms for processing multiprogrammed or multithreaded workloads. Each microthread can process an independent subset of the workload, an effective throughput-enhancing technique. In addition, a processor cluster can reduce the latency of a multithreaded program by running threads on separate processor units concurrently.

Single-threaded programs typically execute in a single microthread. Compilers have difficulty parallelizing most single-threaded programs (except for a restricted class of numerical applications) to generate multiple threads. So, unless a programmer manually threads a program, it won't benefit from the availability of multiple microthreads. Speculative multithreading addresses this problem.[11] A single-threaded program is speculatively threaded at loops or function calls even though dependencies may exist between the threads. Many proposed CMP systems have additional hardware to support this speculation efficiently. There may be data dependencies between threads both in program memory and in register values. Register values can be shared through memory and explicit synchronization to model the data's

Prefix	Size(s)*	No. of instructions	Description	Variations**
OR	W	1	Or	
PACK	D	1	Pack	
PAUSE		1	Pause	
PDIST	B	1	Pixel distance	
PICKC		1	Pick conditional	
POPCOUNT		1	Population count	
POWER	X	1	Parallel power	par
PREFETCH		1	Prefetch data from memory	
RECSQRT	X, SP	2	Reciprocal square root	par
RETRY		1	Return to trapping instruction	
SETHI	H	1	Set high halfword	
SETIR		1	Set internal register	
SETLO	H	1	Set low halfword	
SH	H, W, D	11	Shift left/right logical/arithmetic	par, sat
SIR		1	Software-initiated reset	
SOFTTRAP		1	Software-initiated trap	
ST	B, H, W, D	12	Store register(s) to memory	alt
STC	B, H, W, D	13	Store register(s) to memory conditionally	alt, par
STI	W	1	Store register to instruction memory	
SUB	H, I, L, SP, DP	14	Subtract	par, rnd, sat
SWAP	W	1	Swap register with memory	
TRAPC		1	Trap conditional	
XOR	W	1	Exclusive OR	
YIELD		1	Yield processor unit	
Total		245		

*Values indicate size of the source operands, not the size of the result. Included where applicable. Key: I = integer; L = long integer; SP = single-precision floating-point; X = fixed point; B = byte; H = halfword; W = word; D = doubleword; G = group; U = unsigned.

**Variations include the basic instruction as described as well as combinations of the options described here. Key: alt = alternate address space identifier 1 or 2; cond1 = Z or NZ (zero, not zero); cond2 = EQ, LT, LE, ULT (equal, less than, less than or equal, unsigned less than); cond3 = unsigned low or signed/unsigned high result; cond4 = normal, fast, or speculative; NF = nonfaulting; par = parallel (SIMD for both 32- and 64-bit architectural register widths); pred = prediction (either static or dynamic); rnd = RN, RZ, RL, RH (round toward nearest, round toward zero, round low, round high); sat = with saturation; s/us = signed or unsigned.

producer-consumer relationship. However, synchronization can be expensive in large CMP systems. Thus, most proposed CMPs have a dedicated bus to pass these register values correctly between threads.

MAJC speeds single-threaded-program execution by using speculative multithreading on multiple microthreads. The architecture provides virtual channels (see “How virtual channels work” sidebar on the next page) between different microthreads in a processor cluster to communicate shared register values with producer-consumer synchronization. A speculative thread may have a register read-after-write dependency on a nonspeculative or less speculative thread executing in a different microthread. Code in the nonspeculative or less speculative thread forwards the register value to the speculative thread through a virtual channel. This allows speculative threads to maintain the sequential semantics of the original single-threaded program while still executing somewhat in parallel. However, an implementation or the compiler still must perform memory disambiguation to maintain memory data dependencies between the nonspeculative and speculative threads. The virtual channels are accessible through load and store operations and do not have direct access to the MAJC architectural registers.

The architecture also provides fast interrupts between microthreads in a processor

How virtual channels work

A demonstration of the use of virtual channels illustrates their semantics and utility. This example comes from the SPEC CINT95 benchmark, 129.compress. The following pseudocode represents the original program segment:

```
for (InCnt = value; InCnt > 0; InCnt--) {
    c = {value from buffer, InBuff};
    fcode = f(ent,c);
    i = f(ent,c);
    found = {true if fcode exists in global array htab starting at index i};
    if (found) ent = codetab[ {index where found} ];
    else {
        output(ent); /* Uses ent and free_ent */
        ent = c;
        if (free_ent < 65536) {
            codetab[i] = free_ent++;
            htab[i] = fcode;
        } else ...
    }
}
```

A compiler can determine that memory operations in the procedure *output* do not have dependencies on memory operations in the global arrays *htab* and *codetab* in the rest of the loop. It can also determine that *output* uses the local variable *ent* and the global variable *free_ent*. Two microthreads can speed this benchmark's execution. The microthreads execute the same program segment as follows:

Microthread A: (virtual channel producer)

```
for (InCnt = value; InCnt > 0; InCnt--) {
        c = {value from buffer, InBuff};
        fcode = f(ent,c);
        i = f(ent,c);
        found = {true if fcode exists in global array htab
                starting at index i};
        if (found) ent = codetab[ {index where found} ];
        else {

            /* Send ent to µthread B on ent_channel */
            Send(ent_channel, ent);
            /* Do the same with free_ent */
            Send(free_ent_channel, free_ent);

/* Continue loop iterations while µthread B runs */
            ent = c;
            if (free_ent < 65536) {
                codetab[i] = free_ent++;
                htab[i] = fcode;
            } else ...
        }
}
flush(ent_channel, free_ent_channel);
done = 1; /* Compiler-introduced variable */
```

Microthread B: (virtual channel consumer)

```
while (1) {
Label:
    /* Without blocking, receive lcl_ent */
    /* from µthread A on ent_channel */
    lcl_status = ReceiveNB(ent_channel, lcl_ent);
    if (done && (lcl_status == 0)) break;
    if (lcl_status == 0) {goto Label};

    output(lcl_ent);
    /* Above procedure contains similar */
    /* mechanism to receive lcl_free_ent */
    /* from µthread A  on free_ent_channel */
}
/* Finish execution of µthread B */
```

The compiler must insert code to pass the values of *ent* and *free_ent* from the microthread executing the main loop (Microthread A) to the microthread executing the procedure *output* (Microthread B) using virtual channels. The virtual channels' producer-consumer synchronization ensures that the conditional execution of Microthread B satisfies Microthread A's control dependency on it (that is, Microthread B's dependency on the value of *found*), as follows: Microthread B waits for data on a virtual channel, and Microthread A does not send data on a virtual channel unless *output*, and thus Microthread B, should execute. When the loop is completed, Microthread A notifies Microthread B to end its execution by asserting the compiler-introduced variable *done*. Overall, through virtual channel use, a program that would have executed in t_1 seconds can theoretically execute in t_2 seconds, where t_2 equals t_1 minus the time required to execute the procedure *output*.

cluster so that threads running on different microthreads can quickly notify each other of pertinent events. This fast notification enables speculative threads to be generated and rolled back quickly, reducing the overhead of thread speculation. In addition to their uses for speculative multithreading, virtual channels and fast interrupts benefit multithreaded programs that require producer-consumer data sharing and fast communication.

Virtual channels allow easy maintenance of register dependencies between a nonspeculative thread and speculative threads. A MAJC implementation or compiler still must maintain the correct memory data dependencies between a nonspeculative thread and speculative threads as if they had executed in sequential program order. For typical C or C++ programs, the implementation may have to provide fairly complex hardware to detect and maintain data dependencies between memory operations from each thread. Or the compiler may have to insert significant additional code to check for memory data dependencies between threads if it cannot disambiguate memory operations. Additionally, store operations from a speculative thread must not be committed until the thread becomes nonspeculative. Thus, the implementation must buffer a speculative thread's store operations, or the compiler must version the variables that the store operations update.

Java programs have properties that reduce the overhead of software-based checks for memory data dependencies and versioning of variable updates from speculative threads. These properties allow speculative multithreading of single-threaded Java programs on the MAJC architecture, without complex hardware and large software overhead, using a technique called space-time computing (STC).

STC takes advantage of Java's bytecode properties,[12] which allow runtime software to differentiate between memory operations to the stack and to the heap. STC ensures that variables shared between a nonspeculative thread and speculative threads are shared only through the heap. It achieves this by moving variables from the stack to the heap, if necessary, when speculatively threading at loop boundaries. Variables—except for that containing the predicted return value—are never shared through the stack when STC speculatively threads across method boundaries. With STC, the checks for memory data dependencies between threads are necessary only on memory operations to the heap. The number of memory operations to the stack is usually significantly larger than the number of operations to the heap. Very little code is added to the nonspeculative thread for memory-dependency checking. STC also takes advantage of the Java exception and monitor synchronization model to make the speculative thread as efficient as the nonspeculative thread, even with memory-dependency checking.

The Java stack's state is known at the launch of a speculative thread. The speculative thread uses its own stack for its execution and copies data from the launching thread's stack one frame at a time as needed. Variables in the heap that are updated by a speculative thread are versioned with code in the thread until it becomes nonspeculative. The nonspeculative thread operates in current time and in current memory space. The speculative threads operate in their own separate memory space and future time. When a speculative thread becomes nonspeculative, its space and time dimensions collapse into the current space and time dimensions—hence the term *space-time computing*.

A system with two threads—a nonspeculative head thread and a speculative thread—illustrates the use of STC for speculative multithreading of a single-threaded Java program. We use this system here to explain the STC operation concisely; it can easily be extended to support a greater number of speculative threads. Also, we have omitted a variety of optimizations for clarity.

During a load to a heap variable, the speculative thread updates the status information associated with the variable before loading to its version of the variable. The variable's status information can be kept with the object or array that contains the variable and can be shared with other variables. Also, during a store to a heap variable, the speculative thread updates the variable's status information before determining if it has versioned the variable. If the variable has not been versioned, the speculative thread creates a version for itself. It then stores to its version of the variable. It also maintains a versioned-objects list.

During a store to a heap variable, the head thread updates all versions of the variable.

This allows loads from the variable in the speculative thread to receive the forwarded value from the head thread if the loads were not performed before the store in the head thread. The head thread also checks the variable's status information to determine whether the speculative thread has violated a memory dependency on the variable. If so, the head thread causes the speculative thread to roll back so that the memory dependencies of the program are maintained. Then, the speculative thread uses the versioned-objects list to destroy its versioned variables.

As mentioned earlier, MAJC assumes the release consistency memory model. The speculative thread requires that the store to a variable's status information be ordered with respect to the load to that variable. The head thread also requires that stores to all a variable's versions be ordered with respect to the load from the status information. Release consistency does not order loads and stores to different memory locations, so architectures typically need memory barrier instructions to provide the ordering. Memory barrier instructions can have a significant performance penalty because they order more than the required loads and stores. In contrast, MAJC orders only the necessary loads and stores, using special load and store flavors.

The widespread adoption of the Java platform and the need to support broadband-enabled applications gave us a great opportunity to rethink the design of microprocessors and computer systems. MAJC represents our attempt to take a broad new look at parallelism. It uses microarchitecture techniques and compiler optimizations to exploit instruction-level parallelism. MAJC-based systems will deliver even more application performance by exploiting thread-level parallelism and leveraging the characteristics of high-level languages such as Java.

A well-constructed architecture should form the foundation of state-of-the-art implementations that can be produced by design teams of less than 100 people. Otherwise, implementations tend to become too complex, resulting in

- more staff-years needed for physical implementations,
- more staff-years needed for developing and optimizing compilers and system software,
- more power needed to achieve high clock frequencies (due to expensive and exotic circuits), and
- management difficulties in maintaining effective communication with a large team (potentially leading to staff retention problems).

Starting in early 2001, we hope to demonstrate, through implementations that radically vary in their architectural parameters, such as issue width and number of processor units, that the MAJC architecture yields excellent products implemented by small teams. MICRO

Acknowledgments

We could not have developed MAJC without the close involvement of the implementation team. Many of the features described here originated from the processor, multimedia, compiler, and operating system teams. Only close collaboration between these teams and the architecture team could yield a successful product (fortunately, we all work on the same floor). We thank Bill Joy for his innovative contributions and Andre Kowalczyk for his relentless drive to deliver the first MAJC chip.

References

1. J. Turley, "Alpha Runs x86 Code with FX!32," *Microprocessor Report*, Vol. 10, No. 3, Mar. 5, 1996, pp. 11-13.
2. V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A Transparent Runtime Optimization System," *Proc. ACM SIGPLAN 2000 Conf. Programming Language Design and Implementation,* ACM Press, New York, 2000, pp. 1-12.
3. Sun Microsystems, *Java Hotspot Virtual Machine*, http://www.sun.com/software/communitysource/hotspot/faq.html.
4. Intel Corp., *The IA-64 Architecture Software Developer's Manual Vol. 3 rev. 1.1: Instruction Set Reference*, http://developer.intel.com/design/ia-64/manuals/index.htm.
5. K. Diefendorff et al., "AltiVec Extension to PowerPC Accelerates Media Processing," *IEEE Micro*, Vol. 20, No. 2, Mar.-Apr. 2000, pp. 85-95.

6. B. Smith, "The Architecture of HEP," *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, MIT Press, Cambridge, Mass., 1985, pp. 41-55.
7. R. Alverson et al., "The Tera Computer System," *Proc. Int'l Conf. Supercomputing*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1990, pp. 1-6.
8. K. Olukotun et al., "The Case for a Single-Chip Multiprocessor," *Proc. Seventh Int'l Symp. Architectural Support for Parallel Languages and Operating Systems*, ACM Press, New York, 1996, pp. 2-11.
9. K. Gharachorloo et al., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proc. 17th Int'l Symp. Computer Architecture*, IEEE CS Press, 1990, pp. 15-24.
10. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley, Reading, Mass., 1996.
11. G.S. Sohi, S. Breach, and T.N. Vijaykumar, "Multiscalar Processors," *Proc. 22nd Int'l Symp. Computer Architecture*, IEEE CS Press, 1995, pp. 414-425.
12. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, Reading, Mass., 1996.

Marc Tremblay holds the title Distinguished Engineer at Sun Microsystems, where he is chief architect of the Processor and Products Group, which handles Sparc and MAJC products. He is responsible for creating and implementing microprocessors tailored to the Java computing environment, extreme processing, and broadband services. Tremblay received an MS and a PhD in computer science from UCLA and a BS in physics engineering from Laval University in Canada. He holds 28 patents, with 60 more outstanding, in various areas of computer architecture. He was the cochair of the Hot Chips 2000 conference. He is a member of the IEEE and the ACM.

Jeffrey Chan is a member of the MAJC architecture team. He was the graphics architect for the MAJC 5200 microprocessor. He also worked on various parts of the microarchitecture and the implementation of the MAJC 5200 microprocessor. His research interests are high-performance computer architecture, distributed computing, and graphics and multimedia algorithms. Chan received a BS and an MS in electrical engineering from Stanford University. He is a member of the IEEE and the ACM.

Shailender Chaudhry is a member of the MAJC architecture team. Previously, he worked on the PicoJava-II and MicroJava architectures. His research interests include computer systems architecture, algorithms, and protocols. He was the principal investigator of the space-time computing (STC) technique discussed in this article. Chaudhry received a BS and an MS in computer engineering from Syracuse University and has recently submitted his PhD dissertation on guaranteed quality-of-service systems.

Andrew Warren Conigliaro works at Sun Microsystems, where he currently focuses on MAJC hardware simulation, implementation definition, and microprocessor hardware bringup. His technical interests are high-performance and parallel-computer architectures and simulation analysis. Conigliaro received his BS in electrical engineering from Stanford University and is a member of the IEEE.

Shing Sheung Tse is an engineer at Sun Microsystems, where he has worked on the MicroJava and MAJC microprocessors. His research interests are computer systems architecture and simulation. He received his BS in electrical engineering and computer science from the University of Wisconsin at Madison and his MS in electrical engineering from Stanford University. He is a member of the IEEE.

Direct comments about this article to Marc Tremblay, Sun Microsystems Inc., 901 San Antonio Rd., MS USUN03-202, Palo Alto, CA 94303; marc.tremblay@sun.com.