

A JAVA ILP Machine Based on Fast Dynamic Compilation

Kemal Ebcioglu

Erik Altman

Erdem Hokenek

IBM T.J. Watson Research Center

Yorktown Heights, NY 10598

{kemal,erik,hokenek}@watson.ibm.com

Keywords: JAVA PROCESSORS, INSTRUCTION-LEVEL PARALLELISM, OBJECT CODE COMPATIBLE VLIW, DYNAMIC COMPILATION, BINARY TRANSLATION, SUPERSCALAR

1 Introduction

The **JAVA** programming language has recently been attracting attention because of its desirable properties such as portability (platform independence) and security in the network computing environment. As network computing and **JAVA** gain importance, dedicated **JAVA** processors designed for running **JAVA** programs efficiently (e.g. as proposed by Sun [13]) seem to offer new opportunities in the computing industry.

One application for **JAVA** processors is the cost-sensitive embedded computing market (handheld telephones that support email, palmtop computers, television set-top boxes, and so on), where a **JAVA** processor can be valuable due to its small code size capability.

For the desktop machines for network computing (called Webtops), one needs high performance execution of **JAVA** programs, since these machines will be expected to run substantial applications that will be written in **JAVA**, such as spreadsheets, presentation tools, word processors, and database tools. On the Webtop, a chip designed to run only **JAVA** programs along with a lean operating system to support it, can theoretically provide a cheaper and faster solution compared to a general purpose PC.

2 Objections to JAVA Processors

JAVA processors have engendered great controversy in the industry and academia. See, for example, the arguments from Microsoft against **JAVA** processors and network computers [8].

One main objection against **JAVA** processors has been that there is not enough software written in **JAVA**, and that building operating systems and applications from scratch in **JAVA** can require enormous programming effort. While we agree with the difficulty of developing complex software, given the momentum and significant effort the industry is putting behind **JAVA**, we believe it is only a matter of time before substantial useful applications and complete operating systems will become available in **JAVA**. (Some are already available.)

Another objection has been that language specific processors (such as the *Symbolics Lisp Machine*, or the *Forth Stack Machine*) have not been performance competitive against efficient implementations of these languages on general purpose RISC processors, using state of the art compiler optimizations. While we think that this observation is true, we believe it involves a fundamental misconception about how **JAVA** processors should be designed. Although it is possible to design **JAVA** processors that directly execute byte code in hardware (as Sun seems to propose), for high end implementations of **JAVA** machines we feel it is more appropriate to have the architecture look like an ILP machine with RISC-like primitives, where byte codes are translated under the covers to the RISC primitives, optimized and scheduled by a state of the art just in time (JIT) compiler, and then executed efficiently, thus obtaining the benefits of both the advanced compiler optimizations and ILP techniques. Thus, a stack machine is not needed for executing **JAVA** byte codes.

A third objection against **JAVA** processors has been that there are legacy applications and operating systems that are written for the *x86*, *S/390*, *PowerPC*, or *SPARC* architectures that we cannot stop using all of a sudden. We cannot port all this legacy software to **JAVA** either. This observation of course is also true, and must be addressed by any proposed **JAVA** computing environment. As a solution, Sun has recommended making the new **JAVA** Webtops and some older processors co-exist on the same intranet network, where the older processors can continue to run legacy applications [10]. In this paper, we will propose a new way of implementing legacy code on the same machine that runs **JAVA**, without hardware complexity.

3 Possible Approaches to Designing JAVA Processors

The simplest **JAVA** processor is the stack machine executing byte codes directly, at 1 cycle per instruction for most simple operations. The Sun PicoJAVA machine [13], and the Patriot Scientific Corporation ShBoom processor [9], are in this category. This approach has the advantage of a small code size. However, the stack machine has to execute extra push and pop operations between local variables and the stack, whereas in properly optimized RISC code there are no such push and pop operations. Thus we suspect that a **JAVA** byte code stack machine will not be performance competitive with optimized RISC code running on a simple single integer unit RISC, in the same frequency (barring the ICache advantage of the **JAVA** machine due to its smaller code size).

The example in Figure 1 contrasts a generic **JAVA** machine running byte code directly at a rate of 1 **JAVA** instruction/cycle, vs. a generic single ALU RISC running optimized code at a rate of 1 RISC instruction/cycle, vs. a multiple ALU ILP machine (a VLIW or in-order superscalar). Because the example is short, only two ALU's and a few registers were used in the ILP machine code. As the code fragments get larger, there will be a benefit in using more ALU's and registers in the ILP machine. Note that the performance effect of the two steps outlined in Figure 1 is multiplicative: the performance of the ILP machine is the product of **JAVA** byte code instructions per RISC instruction, and RISC instructions per cycle. Better traditional RISC compiler optimizations can increase the former factor, whereas better ILP compilation techniques and architectural features can increase the latter.

The code generated by just in time **JAVA** compilers is still not as good as optimized RISC code produced by C compilers in terms of code size and performance. Some JIT compilers are even resorting to implementing the **JAVA** stack literally, for example translating a **JAVA** `iadd` operation into loading the top two elements of the stack, adding them and storing the result back onto the stack. We feel this deficiency in JIT compilers is temporary, and that JIT compilers will eventually obtain most of the performance of fully optimized C code (see for example, the Animorphic [2], and Caffeine [7] projects, for efforts already underway for generating better code from **JAVA**, although these are not quite JIT

JAVA Stack Machine =====	Single ALU RISC =====	ILP Machine =====
params=v1,v2,v3,v4 v5=(v1+v2)-(v3+v4) return v5	params=r4,r5,r6,r7 return value=r3 return addr=link reg	(VLIW's separated by *'s)

PROC sub		*****
iload_1	add r3=r4,r5	add r3=r4,r5
iload_2	add r4=r6,r7	add r4=r6,r7
iadd	sub r3=r3,r4	*****
iload_3	b lr	sub r3=r3,r4
iload_4		b lr
iadd		*****
isub		
istore_5		
iload_5		
ireturn		

10 bytes	16 bytes	16 bytes+encoding ovh
10 cycles	4 cycles	2 cycles
1 JAVA IPC	2.5 JAVA IPC	5 JAVA IPC

Figure 1: Code for **JAVA** byte code machine, simple RISC machine, and an ILP machine.

compilers). In this paper, we will also describe a fast just-in-time algorithm to generate optimized RISC code for an ILP processor from **JAVA** byte codes, where both the stack and local variables have been allocated in registers. This algorithm is applicable to ordinary RISC processors such as *PowerPC* as well.

As it is demonstrated in this example, there can be many **JAVA** operations (e.g. `iload`, `istore`) that push local variables onto the stack, and pop the stack top into a local variable. Technically, assuming the stack and local variables are allocated into registers, these push and pop operations are copy operations between registers that do not perform any computation, and can therefore be done in zero time by proper bypassing and data forwarding. So the next higher performance **JAVA** processor would be one that executes push and pop operations between local variables and the stack in zero time. The PicoJAVA processor is able to execute the push operation for the second operand of a two operand stack operation in zero time (e.g. in the context `iload_1`, `iload_2`, `iadd`, `istore_3`, the second `iload_2` takes zero cycles). Improving the operation-collapsing capabilities [12] of a **JAVA** engine would bring the performance of the **JAVA** engine closer to that of the single ALU RISC engine, while keeping the smaller code size advantage.

A third possible approach is to dynamically translate the byte code into RISC primitives each time they are fetched from memory, and issue the RISC instructions to an out of order superscalar pipeline, committing the results in the original order. This is similar to the AMD *K5* superscalar design [1], which converts *x86* instructions into RISC primitives and issues them out of order. Assuming the push and pop operations are converted to register copy operations and therefore are not really issued to functional units (as a good superscalar should do with register copy operations), this approach has

the potential to get the performance of what an equivalent C program gets on RISC superscalars with multiple ALUs. The disadvantage is that such a design would require substantial effort, and a long time to market, comparable to full scale out of order superscalar designs for mainline processors.

An intermediate approach is to convert byte codes to RISC primitives at ICache miss time, and then save the converted code in the ICache, possibly after scheduling [5]. This approach, since it eliminates the dynamic decoding work, results in a shorter pipeline and therefore less branch misprediction penalties. Also, since cache misses are rare, a slower ICache miss processing algorithm can be tolerated. However, the hardware mechanism for doing the conversion from byte codes to the internal ILP form is still complex and requires substantial design effort. Also dynamic translation by hardware does not allow the sophisticated optimizations one could achieve by software.

4 Our Approach

Our approach is to design a **JAVA** processor based on incrementally compiling (e.g. on a page basis), **JAVA** byte codes into a simple internal ILP architecture (a VLIW), and saving the ILP code in a memory area. The next time the same fragment of code (such as a page) is executed, the translated code can be re-used (unless cast out). Any code branching out of its fragment (page) first checks whether the translation of the next fragment is available before jumping to it. When the translation is not available, the incremental compiler is invoked to translate, after which the newly translated code is jumped to.

This approach results in a simpler design, compared to an out of order superscalar. While dynamic compilation to an internal representation is likely to require more memory than hardware that executes byte codes directly, the code size advantage of **JAVA** can be mostly preserved in our architecture, by interpreting the less frequently executed **JAVA** code, while reserving the dynamic compilation to the frequently executed parts of the **JAVA** programs.

Our approach obtains **JAVA** performance in three tiers:

1. When a byte code fragment is first executed, the byte codes are converted into optimized RISC primitives. This is done by laying out the stack, and local variables in symbolic registers. The example in Figure 2 demonstrates the translation of **JAVA** into intermediate RISC code with symbolic registers.
2. These optimized RISC primitives are then executed at a rate of multiple operations per cycle, through static scheduling done concurrently with register allocation. Register copy operations (resulting from push and pop operations between the stack and local variables) are coalesced away, generating no code at all.

The RISC operations are processed in the order they appear in the code, as shown in Figure 3. $h(v)$ indicates the current mapping of symbolic register v to a hardware register. A copy operation merely makes the mapping of the destination equal to the mapping of the source. An operation other than a copy is greedily scheduled in the first possible VLIW using the current mappings of the symbolic source registers, and an appropriate hardware register is chosen for the destination symbolic register, after taking into account which hardware registers have become free, due to last uses of symbolic registers. There are some mandatory register mappings, for instance the symbolic parameter registers, $p0, p1$, must be mapped to $r3, r4, \dots$, and the return value must be placed in $r3$, as required by *PowerPC* linkage conventions.

Byte Code	Intermediate Code with Symbolic Registers	
=====	=====	
PROC sub	COPY v1=p1	(p0, p1, p2,... are parameters)
	COPY v2=p2	(v1, v2,... are local variables)
	COPY v3=p3	(s1, s2,... are stack locations)
	COPY v4=p4	
iload_1	COPY s1=v1	
iload_2	COPY s2=v2	
iadd	ADD s1=s1,s2	
iload_3	COPY s2=v3	
iload_4	COPY s3=v4	
iadd	ADD s2=s2,s3	
isub	SUB s1=s1,s2	
istore_5	COPY v5=s1	
iload_5	COPY s1=v5	
ireturn	COPY p0=s1	(p0 is return value register)
	B lr	(lr contains return address)

Figure 2: Byte code and initial translation to ILP intermediate code.

The final ILP code is the same as column 3 of Figure 1. (Two VLIWs for the entire program). Below we describe the scheduling algorithm in more detail, as well as the generation of spill code when needed.

3. The architecture has support for graphics vector operations similar to the *VIS* instruction set from Sun, or the *MMX* instruction set from Intel. Thus graphics library routines called by **JAVA** can execute with more performance.

An advantage of this particular incremental compilation approach is that the design is scalable. The same incremental compiler software can be used to run on 2,4,8, or 16 ALU machines without changes, since compilation is dynamic.

4.1 Scheduling Algorithm

We now fill in some of the details of our scheduling algorithm, along with our mechanism for spilling registers. Figures 4 and 5 depict the scheduling algorithm. Contrary to the topic of this workshop, the algorithm is depicted in a mix of C and English. To keep the flow clear and reduce clutter, the algorithm does not show handling of “corner cases” such as bytecodes that span a page (or other fragment size) boundary, although they are straightforward. Some data structures are also simplified for clarity.

Lines 1-3 in Figure 4 create an empty VLIW instruction, *root_vliw* with no predecessor and a schedule time of 0. The algorithm builds a set of paths from this *root_vliw*, with the set of paths forming a tree. Fork points in the tree are conditional branch bytecodes.

The set of VLIW instructions coming from a single root instruction form a *group*. At any particular time, the algorithm may be scheduling multiple groups. This is reflected in lines 5-6 which initialize *vliw_group_list* to have a single group with *root_vliw* and the *entry_bytecode* to the **JAVA** method starting instruction.

Byte Code	Intermediate Code with Symbolic Registers	Scheduling Actions
=====	=====	=====
		Start:
		h(p0)=r3, h(p1)=r4, h(p2)=r5
		h(p3)=r6, h(p4)=r7
PROC sub	COPY v1=p1	h(v1)=h(p1)=r4
	COPY v2=p2	h(v2)=h(p2)=r5
	COPY v3=p3	h(v3)=h(p3)=r6
	COPY v4=p4	h(v4)=h(p4)=r7
iload_1	COPY s1=v1	h(s1)=h(v1)=r4
iload_2	COPY s2=v2	h(s2)=h(v2)=r5
iadd	ADD s1=s1,s2	s1, s2 are ready in VLIW 1
		s1 is last use, free h(s1)=r4
		s2 is last use, free h(s2)=r5
		Assign r3 (first free reg) to dest s1,
		New h(s1)=r3
		==> Schedule ADD r3=r4,r5 in VLIW 1
iload_3	COPY s2=v3	h(s2)=h(v3)=r6
iload_4	COPY s3=v4	h(s3)=h(v4)=r7
iadd	ADD s2=s2,s3	s2,s3 are ready in VLIW 1
		s2 is last use, free h(s2)=r6
		s3 is last use, free h(s3)=r7
		New h(s2)=r4, First Free Reg
		==> Schedule ADD r4=r6,r7 in VLIW 1
isub	SUB s1=s1,s2	s1 and s2 are ready in VLIW 2
		s1 is last use, free h(s1)=r3
		s2 is last use, free h(s2)=r4
		Preferred Assignment for s1 is r3
		Assign New h(s1)=r3
		==> Schedule SUB r3=r3,r4 in VLIW 2
istore_5	COPY v5=s1	h(v5)=h(s1)=r3
iload_5	COPY s1=v5	h(s1)=h(v5)=r3
ireturn	COPY p0=s1	h(p0)=h(s1)=r3
	B lr	==> Schedule B lr in VLIW 2

Figure 3: Byte code and processing of ILP intermediate code.

When there are multiple groups, it is important to schedule from the most likely to execute path first. This is accomplished on line 10, which reflects the highest priority group in `curr_vliw` and `bytecode_ptr`. `curr_vliw` is the last VLIW instruction on some path from the root instruction of its

group. The VLIW instructions on a path are chained together by pointing to their predecessors. This is reflected in the 0 parameter to `new_vliw ()` on line 2. The parameter to `new_vliw ()` represents the preceding VLIW instruction on the current path, and is NULL for the first VLIW instruction. `bytecode_ptr` points to the next bytecode to be scheduled along the current path.

Branches that go beyond the current page will be translated to a call to the translator, as shown in lines 11-14. If and when this off-page branch is executed for the first time, the translator will generate new native VLIW code for the target page entry point.

Line 16 begins a `do-while` loop which scans through all the bytecodes in the current basic block, converting them to RISC operations, and then scheduling the RISC operations. Line 17 obtains the current bytecode, `b_op` and then increments the pointer to the next bytecode. Line 19 translates `b_op` to RISC ops, while line 21 steps through the translated RISC ops, `risc_op`.

Lines 21-33 represent the core of the algorithm, specifically how the RISC ops are scheduled into VLIW instructions. The first step (line 23) is `invoke_min_risc_op_time ()`, which is depicted in Figure 5. `min_risc_op_time ()` determines the earliest that data dependences permit `risc_op` to be scheduled. (Control dependences can be ignored — if `risc_op` is speculatively scheduled above a control dependence, the result register is appropriately renamed. The **JAVA** virtual machine maintains exceptions properly through traps (e.g. range checks); these traps are not moved above branches or other traps or stores.)

The next step on line 19 is to make a `vliw_list` of VLIW instructions on the current path, with the first instruction being the VLIW at time `earliest` and the last being `curr_vliw`. (The `.prev` links of the VLIW instructions are used in constructing this list.) The `for` loop on lines 26-29 steps through `vliw_list` looking for an instruction in which `risc_op` will fit — meaning that a function unit is available, as well as a hardware register for the result. The hardware register will be a register that is not live anywhere on the path from `vliw` to `curr_vliw`. If `vliw` meets these requirements, the loop is exited. (The `fits_vliw ()` function is depicted in Figure 5.)

Line 31 rechecks whether a suitable VLIW instruction was found for `risc_op`. If not, the current path is extended by creating a new VLIW instruction and changing `curr_vliw` to it, as well as assigning `vliw` to the new `curr_vliw`. After this check and possible update, `risc_op` is finally scheduled in the appropriate VLIW instruction, as represented by `vliw`.

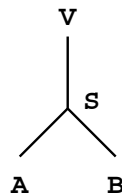
Once the all RISC ops for the current bytecode `b_op` have been scheduled, the algorithm checks whether `b_op` is a branch and if so, what kind.

All branches end the current basic block and hence the outer `while` loop from lines 10-49. In addition onpage, direct branches require that their targets be placed in `vliw_group_list`, which is done on lines 38 and 41. Conditional branches also require that the fall-through branch be added to `vliw_group_list`.

We now outline a method to perform register allocation and spilling.

1. On entry to a procedure the parameters are contained in registers `r3, r4, ..., r11`. This conforms with *PowerPC* parameter passing. The caller will execute, before the call, symbolic register code that copies the symbolic stack top registers (e.g. `s2, s3, ...`) into symbolic parameter registers `p0, p1, ...` which will be always allocated in `r3, r4, ...`. The callee will execute symbolic register code that copies `p0, p1, ...` into symbolic local variable registers `v0, v1, ...`.
2. On return from a procedure the returned result is placed in `r3`. The callee will execute symbolic register code that copies the stack top symbolic register to `p0`, which will be always allocated in `r3`. The caller will then execute symbolic register code that copies `p0` to its stacktop symbolic register.

3. The stack is pointed to by r1 (as required by *PowerPC* linkage conventions). The stack is not the **JAVA** stack. All **JAVA** stack elements and local variables are allocated in registers. The stack is used only for saving and restoring callee-save real registers (r13-r31 in *PowerPC*) and the return address register, if destroyed in a procedure, and for the spill area if needed. Stack frames are linked with a back chain, so an interrupt handler can analyze the call chain for proper interrupt delivery to the JVM. Our stack is similar to the *PowerPC* C stack.
4. Points (1) and (2) are enforced
 - By choosing an appropriate mapping within the group, so that as much as possible, values are in the appropriate registers (e.g. the return value is in r3) upon exit from each group.
 - By using register copy's, where values are not in the proper location at group exit.
5. Since each group of VLIW's forms a tree, optimal and efficient coloring can be done with the "left-edge-algorithm" [11]. The left-edge-algorithm sweeps lifetimes from left to right, with each new lifetime being assigned the first color in the FREELIST, and colors being returned to the FREELIST at the end of each lifetime. Each path through the group can be colored this way in turn, with later paths constrained in their initial paths by the coloring used in earlier paths:



For example, if path VSA is colored first, the coloring on path VSB is constrained in the VS portion by that already done for path VSA. In addition, the coloring should be constrained to have, e.g., the return parameter in r3 at a group exit. For example, if the entire group consists of the VLIW instruction above, this would mean having the proper mapping at points A and B.

In places ("fat spots" [6]) where more values are live than there are physical registers to accomodate them, values can be spilled. Various heuristics could be employed to choose spill candidates. One reasonable choice would be to spill those values with the least references in each fat spot.

5 Dealing with Legacy Code

Initially we were planning to make a pure **JAVA** engine when we started our project. We were planning to extend the **JAVA** virtual machine architecture by defining what happens on powerup reset (machine branches to a fixed location in read only memory containing the boot code), what happens on a TLB miss (an interrupt to a fixed location occurs), how I/O is to be performed (through memory mapped load and store operations to a peripheral bus, such as the PCI bus).

However, we realized that our software prototype task would be simplified if the **JAVA** processor became an extension of a *PowerPC* processor implemented in the same incremental compiler method. We already have a project called **DAISY**(Dynamically Architected Instruction Set from Yorktown) which implements a 100% architecturally compatible *PowerPC* [4]. Notice that since both *PowerPC* and **JAVA** execution is through dynamic translation, and virtual memory is implemented in software

through an exposed TLB, there is not much additional hardware requirement to support a *PowerPC* processor in our design, compared to supporting **JAVA** only.

Here is how the *PowerPC* architecture is extended for direct **JAVA** execution: It requires adding a single new instruction to *PowerPC*:

`Execute_JAVA_Code &save_area`

To use the direct execution feature of our **JAVA** engine, the programmer loads some .class file byte codes into memory, sets up `save_area` to point to the first **JAVA** instruction, and branches to the `Execute_JAVA_Code` *PowerPC* instruction. Then the **JAVA** code starts executing. The `Execute_JAVA_Code` instruction can end in two ways:

1. By an interrupt (such as page fault, or external I/O or timer interrupt). In this case the ILP machine state is saved in the `save_area`, including the program counter. The address of the `Execute_JAVA_Code` *PowerPC* instruction is reported to the *PowerPC* operating system first level exception handler. When the OS is done with the interrupt, it executes a `return-from-interrupt` instruction to the `Execute_JAVA_Code` instruction, which in turn restores the ILP machine state and recommences execution where it left off. The `save_area` is touched before execution, to guarantee that its page will be in memory when needed.
2. By a call to a native *PowerPC* routine. In this case the link register (lr) (return address) is set to the address of the `Execute_JAVA_Code` instruction. The `save_area` contains the ILP machine program counter to restart from, and the ILP machine registers to restore. When the *PowerPC* routine returns to the `Execute_JAVA_Code` instruction, the ILP machine state will be restored and the ILP machine will resume execution.

If the incremental compiler software does choose to translate rather than interpret **JAVA** code, the translated code can be saved in a memory area hidden from the *PowerPC* architecture. The user can change `save_area` inadvertently or maliciously, but can do no more harm than that which can be done with user mode instructions.

A similar technique could be used for embedding **JAVA** in *x86* or *S/390* architectures. In a **JAVA** OS-like operating system most of the time can be expected to be spent in `Execute_JAVA_Code` instructions. We believe keeping the old architecture around is a good safeguard, for being able to run existing code if needed.

6 The Hardware Design

So far we are planning to design for a modest 8 functional unit ILP machine design on a single chip. The design is based on tree VLIWs, as described in [3]. The chip interface will be compatible with the *PowerPC* 6XX bus pinout specification. We envisage our chip as part of a single board network computer, consisting of our *PowerPC/JAVA* chip, an L2 cache, EDO DRAM, and *PowerPC* boot ROM, and a 6XX-PCI bus bridge connected to various PCI peripherals on the board (serial I/O for a keyboard and mouse, graphics and monitor connection, fast Ethernet, optional hard disk.) We plan to place the incremental compiler in previously unused areas of the *PowerPC* boot ROM, and store the translations of *PowerPC* code in the upper 4M (size can be varied by software) of system memory. So the *PowerPC* operating system will see a little less memory than is really available. Updates to the incremental translator and **JAVA** JIT/interpreter can be downloaded from the network. At the end of this paper, we show a preliminary functional organization, as well as a preliminary floorplan for a chip implementing our approach.

We are in the process of embedding the **JAVA** incremental compilation as part of our **DAISY** incremental translation framework for *PowerPC* [4]. We will report on our future progress as it becomes available.

7 Conclusions

We have described a software and hardware design with the following features:

1. High **JAVA** performance is achieved by simple hardware using three tiers for parallelism:
 - (a) A good JIT compilation technique that maps the stack and local variables onto registers where possible.
 - (b) Multiple operations are executed concurrently in each cycle through state of the art ILP compilation techniques.
 - (c) Graphics support hardware improves performance further for library routines.
2. A mechanism to embed **JAVA** execution inside the *PowerPC* architecture, that allows compatibility with legacy *PowerPC* operating systems.

We feel that our incremental translation approach is a step toward creating new kind of open system, where multiple operating systems for different architectures can run on the same hardware.

References

- [1] Advanced Micro Devices, *AMD K5 Processor Technical Reference Manual*.
- [2] Animorphic Systems Web Site, <http://www.animorphic.com>.
- [3] K. Ebcioglu, *Some Design Ideas for a VLIW Architecture for Sequential-Natured Software*, In Parallel Processing (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing), edited by M. Cosnard et al., pp. 3-21, North Holland.
- [4] K. Ebcioglu and E. Altman, *DAISY: Dynamic Compilation for 100% Architectural Compatibility*, Research Report RC20538, IBM T.J. Watson Research Center.
- [5] M. Franklin and M. Smotherman. *A Fill-unit Approach to Multiple Instruction Issue*, Proc. MICRO-27, 1994.
- [6] L.J. Hendren, G.R. Gao, E.R. Altman, and C. Mukerji. *A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs*, The Journal of Programming Languages, Volume 1, Number 3, 1993, pages 155-185.
- [7] Cheng-Hsueh A. Shieh, John C. Gyllenhaal, Wen-mei W. Hwu. *JAVA Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results* Proc. MICRO-29, IEEE Press, 1996.
- [8] Microsoft, *Network Computing: Beyond the Hype*. Microsoft Market Bulletin. 1996. See <http://www.microsoft.com/backoffice/beyond.htm>.
- [9] Patriot Scientific Corporation, *The PSC1000 ShBoom Processor*. See <http://www.ptsc.com/shboom/>.
- [10] Bud Tribble, *JAVA Computing in the Enterprise: What it Means for the General Manager and the CIO*. Sun Whitepaper, 1996. See <http://www.sun.com/961029/JES/whitepapers/>.
- [11] Alan Tucker, *Coloring a Family of Circular Arcs*. SIAM Journal of Applied Mathematics, Volume 29, Number 3, November 1975, pages 493-502.
- [12] S. Vassiliadis, B. Blaner, R.J. Eickemeyer, *SCISM: A Scalable Compound Instruction Set Machine*. IBM Journal of Research and Development, vol. 38, no. 1, January 1994.
- [13] P. Wayner, *Sun Gambles on JAVA Chips*. Byte Magazine, November 1996.

```

1. /* root_vliw is the root VLIW ins of a tree of VLIW tree instrucs */
2. root_vliw      = new_vliw (0);
3. root_vliw.time = 0;
4.
5. vliw_group_list = create_group_list();
6. add_group (vliw_group_list, root_vliw, entry_bytecode);
7.
8. /* curr_vliw is the latest VLIW ins on the current path. */
9. /* bytecode_ptr points to the bytecode continuation of the current path */
10. while ((curr_vliw, bytecode_ptr = high_prio (vliw_group_list)) != NULL) {
11.     if (bytecode_ptr is off page) {
12.         Generate call to translator.
13.         bb_end=TRUE; continue;
14.     }
15.
16.     do {
17.         b_op = *bytecode_ptr; bytecode_ptr+= size(b_op);
18.
19.         risc_ops_list = bytecode_to_risc (b_op);
20.         num_risc_ops  = 0;
21.         while (risc_op = risc_ops_list[num_risc_ops++]) {
22.
23.             earliest = min_risc_op_time (risc_op, curr_vliw);
24.             vliw_list = make_vliw_list  (curr_vliw);
25.
26.             for (time = earliest; time < vliw_list.size; time++) {
27.                 vliw = vliw_list[time];
28.                 if (fits_vliw (risc_op, vliw)) break;
29.             }
30.
31.             if (time == vliw_list.size) vliw = curr_vliw = new_vliw (curr_vliw);
32.             vliw.op_list[vliw.op_list_size++] = risc_op;
33.         }
34.
35.         switch (b_op.type) {
36.             case COND_BRANCH:
37.                 add_group (vliw_group_list, curr_vliw, bytecode_ptr);
38.                 add_group (vliw_group_list, curr_vliw, branch_targ);
39.                 bb_end = TRUE;    break;
40.             case DIRECT_BRANCH:
41.                 add_group (vliw_group_list, curr_vliw, branch_targ);
42.                 bb_end = TRUE;    break;
43.             case RETURN_BRANCH:
44.                 bb_end = TRUE;    break;
45.             default:
46.                 bb_end = FALSE;   break;
47.         }
48.     } while (! bb_end);
49. }

```

Figure 4: Main scheduling algorithm.

```

min_risc_op_time (risc_op, curr_vliw)
{
    latest = Latest time any operand of risc_op available;

    switch (risc_op.type) {
        case STORE:
        case COND_BRANCH:
            return max (latest, curr_vliw.time);

        default:
            return latest;
    }
}

fits_vliw (risc_op, vliw)
{
    if (! sufficient_func_units (risc_op, vliw)) return FALSE;
    if (! hardware_reg_for_result (risc_op, vliw)) return FALSE;
    else return TRUE;
}

```

Figure 5: Auxiliary routines for main scheduling algorithm.