DESIGN AND ANALYSIS OF HARDWARE FOR HIGH PERFORMANCE PROLOG

BRUCE K. HOLMER, BARTON SANO, MICHAEL CARLTON, PETER VAN ROY,[1] AND ALVIN M. DESPAIN[2]

▷ Most Prolog machines have been based on specialized architectures. Our goal is to start with a general purpose architecture and determine a minimal set of extensions for high performance Prolog execution. We have developed both the architecture and optimizing compiler simultaneously, drawing on results of previous implementations. We find that most Prolog specific operations can be done satisfactorily in software; however, there is a crucial set of features that the architecture must support to achieve the best Prolog performance. In this paper, the costs and benefits of special architectural features and instructions are analyzed. In addition, we study the relationship between the strength of compiler optimization and the benefit of specialized hardware. We demonstrate that our base architecture can be extended to include explicit support for Prolog with modest increase in chip area (13%) and yet attain a significant performance benefit (60 to 70%). Experiments using optimized code that approximates the output of future optimizing compilers indicates that special hardware support can still provide a performance benefit of 30 to 35%. The microprocessor described here, the VLSI-BAM, has been fabricated and incorporated into a working test system. ◁

This paper is a revised and expanded version of "Fast Prolog with an Extended General Purpose Architecture," *17th International Symposium on Computer Architecture*, 1990. This work was partially funded by the Advance Research Projects Agency (ARPA) and monitored by the FBI under contract No. J-FBI-91-194 and the Office of Naval Research under contract No. N00014-88-K-0579. B. Holmer is currently funded by the National Science Foundation under the Research Initiation Award MIP-9210692.

Address correspondence to Bruce Holmer, EECS Department, Northwestern University, 2145 Sheridan Road, Evanston, IL 60208.

University of Southern California, Denney Research Building, Rm. 17, Los Angeles, CA 90089.

USC Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292.

[1]Programming Systems Lab, DFKI, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany.

[2]EE Systems/ACAL, University of Southern California, 3740 S. McClintock, Los Angeles, CA 90089.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

1. INTRODUCTION

Logic programming in general and Prolog [27] in particular have become popular for rapid software prototyping, natural language translation, and expert system programming. Prolog's use of dynamic typing, backtracking, and unification place heavy computational demands on general purpose computers. In an attempt to achieve ever higher performance, several special purpose architectures have been proposed and built. Early Prolog architectures [37] were microcoded interpreters. Because no compilation was done, performance was disappointing. Higher performance processors [2, 9, 20, 25] have since been based on the Warren Abstract Machine (WAM) [36]. Their instruction sets were derived from the WAM to support execution of Prolog programs. These processors are special purpose, microcoded engines that depend on parallel execution of operations within each relatively coarse-grained instruction for high performance. Initial designs implemented only the instructions that supported the WAM and depended on a host processor for non-WAM computations. To support Prolog built-ins (primitive Prolog operations provided by the system) and system I/O, newer designs incorporate general purpose instructions to minimize dependence on a host. Alternatively, the use of a simple, non-WAM instruction set better supports compiler optimization. Several such special purpose reduced instruction set architectures have been proposed for logic programming [11, 18, 19, 24]. These architectures include primitives that support the use of tagged data, pointer dereference, and multi-way branches. Our hypothesis is that providing support for both compiler optimization and low-level operations can best be accomplished by extending a simple general purpose architecture to support Prolog without compromising the general purpose performance.

The performance improvements of recent general purpose architectures over older architectures can be traced to research in which both the compiler and architecture were developed together [13, 16, 22]. Architectural features that cannot be used by the compiler or that cannot demonstrate performance improvement are not included. Likewise, architectural features are added that support often used primitive operations. We have adopted this approach from the beginning of our project.

It has been conjectured that commercial special purpose symbolic processing architectures are doomed because they are not commodity items, and consequently, economics prevent them from staying on the leading edge of implementation technology. However, if the architectural features necessary to improve symbolic performance are modest and do not interfere with the general purpose architecture, then as more chip area becomes available, future implementations of general purpose processors can deliver high performance symbolic computing in a standard product. We hope that our work is a step towards this result.

The VLSI-BAM design was begun at University of California, Berkeley as part of the Aquarius project. The Aquarius project group relocated to the University of Southern California in 1989 where the VLSI-BAM chip and a custom cache board were completed, fabricated, and tested.

This paper presents the design of a processor based on the Berkeley Abstract Machine (BAM) and motivates its design with the results of our preliminary studies. We also present a discussion of the optimizing compiler, a cost/benefit analysis of the architectural features, and the simulated performance. Section 2 summarizes the processor architecture and hardware implementation. Section 3 presents the instruction set along with the results of our studies which motivated instruction

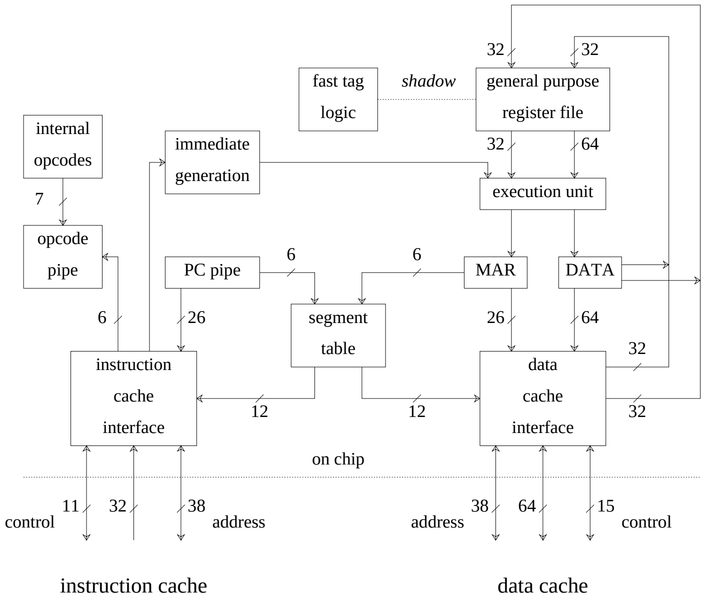

FIGURE 2.1. Block Diagram of the VLSI-BAM Processor

selection. The compilation of Prolog programs is described in Section 4, and in Section 5 we present a cost/benefit analysis of the special features and instructions. Section 6 gives the performance results. The final section concludes with a summary of our results.

2. PROCESSOR ARCHITECTURE AND IMPLEMENTATION

The VLSI-BAM processor is a general purpose, single chip, pipelined processor with extensions to support Prolog execution (Figure 2.1). Both data and instruction words are 32 bits, and most instructions execute in a single cycle. The main features for Prolog are tag manipulation (integrated into arithmetic and the memory system), a double-word data port to memory, special branch on tag support, and several instructions to support our execution model for Prolog.

The architecture is presented in detail along with our motivations in the subsections below. Retaining a core general purpose architecture imposes constraints on the symbolic extensions. For example, the processor should be able to handle tagged data items as single entities, with no special treatment for the tags. We discuss the ramifications of this on the word format and the virtual memory system. Then we present the architecture's register structure and memory interface. Finally, we present some details of the implementation such as the pipeline structure and our mechanism for multiple-cycle instructions.

2.1. Word Format

Prolog does not require the user to specify the type of a data item. This requires that run-time type checking be implemented by adding a tag to each data item to encode the type of that item. Many Prolog processors handle the tag and value fields separately. This approach does not satisfy our goal of integrating tagging into a general purpose architecture. Instead, we use a standard 32-bit word length and place the tag in the most significant four bits of the word. Arithmetic computations and addresses, however, use the entire 32-bit word, so general purpose computations are not affected by Prolog's use of tags. Tag values fixed by the hardware are those for non-negative integers (0000) and negative integers (1111). This selection of tags for integers is a common technique used by Lisp implementations on general purpose machines [26]. We have also fixed the tag value for variable pointers (tvar = 0001) to increase the number of bits available for branch displacements in several Prolog specific instructions.[3] All other tag values are software defined. Our Prolog implementation uses tags similar to those of the WAM.

2.2. Segmented Virtual Addresses

One consequence of using both the tag and value as an address is that each data type is mapped into its own area of virtual memory; however, Prolog's execution model places data with several data types on the same stack or heap. One possible solution is to mask (zero) the tag bits of the address before using it to access memory. This solution is not satisfactory when applied to applications not using tags (for example, C programs). To avoid this difficulty, we have introduced a segment table that maps the most significant six bits of an address to a twelve-bit value (Figure 2.2). An address before mapping is referred to as a short virtual address (SVA), and the 38-bit address resulting from the mapping is referred to as a long virtual address (LVA). This memory segmentation scheme is similar to the segmentation used in the 801 processor [7]. The 801 uses segmentation to extend the virtual address space; however, our primary motivation for using segmentation is to allow multiple data types to be mapped to the same LVA segment. Mapping two bits in addition to the tag bits allows the use of several memory areas for a given data type, each area using a different mapping.[4] At one extreme all data types can be mapped to the same LVA segment (this is equivalent to masking the most significant six address bits). At the other extreme, all SVA segments can be mapped to distinct LVA segments. In our current implementation of Prolog, variable, list, and structure pointers are mapped to the same LVA segment, whereas the environment/choice point stack, the trail stack, and the symbol table are mapped to separate segments.

Another use of segmentation is for sharing data in a multiprocessor system. In this case the 38-bit LVA is used as the global virtual address and sharing of data by cooperating processes is done at the segment level.

2.3. Memory Interface

The high memory bandwidth requirement of Prolog dictates separate instruction and data buses (Figure 2.1). In addition, we have expanded the data bus to double-word width. A double-word data bus is motivated by Carlson's study [5] of the ar-

[3] The *dref*, *uni*, *swb*, and *swt* instructions all assume tvar = 0001 as part of their definition (see Table 3.3).

[4] Although mapping more than six bits may increase the flexibility of this scheme, mapping additional bits would drastically increase the size of the on-chip segment mapping table.

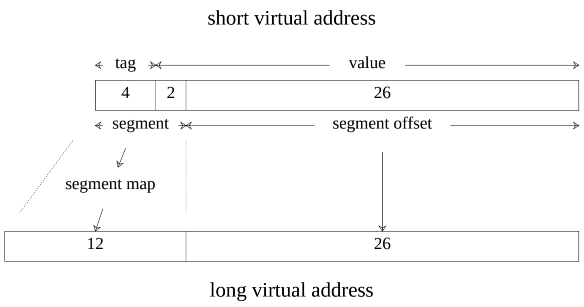

FIGURE 2.2. Segmentation of Virtual Address Space

chitectural requirements of high performance Prolog processors. Carlson compiled Prolog programs into basic register transfer level operations and then compacted them into more complex instructions while enforcing microarchitectural constraints. His results show that the best performance/cost tradeoff occurs when the architecture provides a double-word port to data memory.

A double-word memory port improves the performance of term creation and speeds block transfers to and from environments and choice points. Some previous Prolog processors support fast choice point creation and restoration through the use of specialized buffers or shadow registers [9, 24]. Such hardware solutions are costly and do not fit our goal of maintaining a general purpose architecture. Instead, we rely on double-word memory operations and on compiler optimization to minimize shallow backtracking [33].

Our processor design is tightly coupled with the cache design. We decided against on-chip caches since, in our case, it is more appropriate to use processor chip area for architectural features and use fast, dense static RAM chips for large caches. To speed cache accesses, however, protection violation and consistency checks and address tag comparison are done on-chip. More details about the cache interface are given in [6].

2.4. Base Architecture

All programmer visible processor registers are accessed as two sets of 32 registers: the general purpose register set and the special register set. The general purpose registers are used for procedure argument passing, temporary storage, and as stack pointers. The only general purpose register with a preassigned use is the continuation pointer (r31). This register is implicitly set to the return address by the call instruction. All other uses of the general purpose registers are defined by software convention.

The special registers provide access to the processor status word (PSW), program counter (PC), partial product/quotient register (PQ), segment mapping table, cache interface configuration registers, and a set of fifteen extra registers (s0-s14).

2.5. Implementation Details

The execution pipeline consists of five stages (Figure 2.3). All instructions that modify registers or memory do so in the last pipeline stage. Register bypassing forwards results of the ALU and memory read pipeline stages to instructions in earlier stages of the pipeline. Hardware pipeline stalls are provided in hardware to insure correct execution of both load and store operations. If data from a load instruction is used by the next instruction, then the next instruction is delayed by a cycle. Also, memory instructions immediately following a store are delayed by a cycle. Store instructions require access to the cache during both the M and W pipeline stages—M to provide the address to determine the cache hit/miss and W to provide the data to be stored.

I	instruction fetch
R	register read
A	ALU
M	memory read
W	register/memory write

FIGURE 2.3. VLSI-BAM Processor Execution Pipeline

All instructions are 32 bits with a 6-bit opcode and fixed source register format. Instruction execution is controlled by an opcode pipeline that operates in parallel with the execution pipeline. Each stage of the opcode pipe decodes the opcode associated with that stage of the execution pipeline. Multi-cycle instructions and conditional instructions are implemented using "internal opcodes" [21]. The internal opcodes of multi-cycle instructions are fetched from a PLA and inserted into the opcode pipeline. When an internal opcode is inserted, no instruction is fetched during that cycle. Thus a single external opcode can invoke a sequence of internal opcodes to provide for often used complex operations (for example, pointer dereferencing). Internal opcode insertion is also used for atomic synchronization operations, for pipeline interlock delays, and for trap and interrupt handling. Conditional execution is implemented by conditionally replacing an opcode in the opcode pipe with an internal opcode. Our design uses 55 external opcodes and 24 internal opcodes; of the internal opcodes, nine are related to traps (*trap*, *rft*), 13 implement multi-cycle instructions (*dref, stx, std, pushd, las, jmpr*), and two implement conditional operation instructions (*uni, pusht*).

"Fast tag logic" is used to implement single-cycle tag-compare-and-branch instructions. The fast tag logic consists of an extra register file that duplicates the tag portion of the general purpose register file and special tag comparison logic that allows quick tag comparison and branch. Previous Prolog processors [9] have also duplicated tag bits to accelerate branching on tag value.

The general purpose register file has two read ports (one single-word and one double-word) and two write ports (both single-word). This port structure provides the bandwidth required by single-cycle double-word memory accesses without greatly increasing the complexity of the register file design.

Figure 2.4 is a photomicrograph of the VLSI-BAM chip. The layout consists of three rows of functional units. In the top row, the left-most quarter contains the register index latches, register bypass logic, fast tag comparison logic and fast tag register file. The middle half of the row contains instruction address circuitry that consists of three branch offset adders (two for the three-way branches, *swt*

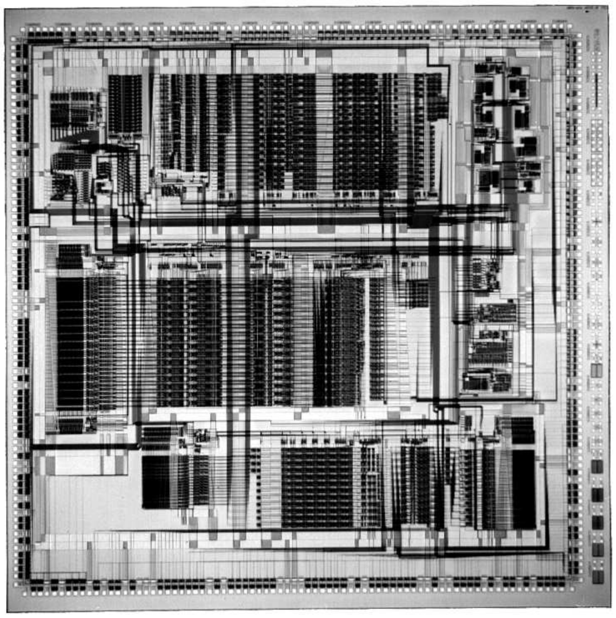

FIGURE 2.4. Photomicrograph of VLSI-BAM Chip Die

and *swb*, and one for *btg*), an incrementer, multiple latches that hold the PC value of each pipeline stage, latches for trap handling, and the partial product/quotient register. The right-most quarter of the top row contains the pipeline control logic that consists of decode PLAs and latches for holding the opcode of each pipeline stage.

The left three quarters of the middle row contains the main data path. The general purpose register file is on the left end and the barrel shifter and ALU are toward the right end and pipeline latches lie in between. The right quarter of the middle row contains condition code logic, logic for trap detection and prioritization, and random logic for pipeline control signals.

At the left end of the bottom row is the special purpose register file. The middle and right end of the row contains the cache interface logic that consists of (moving left to right) the protection specification latches, protection comparison logic, tag comparison logic, instruction and data address latches, and the segment table.

The design is pad limited, so no extra effort was expended to compact the layout beyond what was necessary to fit into the pad frame.

3. INSTRUCTION SET

In this section we present the VLSI-BAM instruction set. The instructions are divided into three groups: general purpose, Prolog inspired general purpose, and Prolog specific. The general purpose instructions are those which can be found in typical processors. The Prolog inspired instructions are those which are not often present in general purpose processors, but which can still be used for general computation. The remaining instructions are tailored specifically to the requirements of Prolog execution.

The general purpose instructions are summarized in Table 3.1. It is important to point out that all arithmetic and logic operations operate on the full 32-bit word. Also, conditional branches consist of separate compare and branch instructions. Compare instructions set or clear the TF (true-false) condition code bit, and the branch instructions take the branch when TF is set. Branches, jumps, and calls are delayed by one instruction. The instruction in a branch delay slot can always be executed (*bt*), annulled (turned into a nop) if the branch is taken (*btat*), or annulled if the branch is not taken (*btan*). Both directions of annulling are included because Prolog often favors annulling when the branch is taken (for example, branching out of straight-line code to the unification failure routine), whereas conditional branches to the top of a loop (common in procedural languages) favor annulling when the branch is not taken.

Special load and store instructions (*ldl* and *stu*) activate a signal sent to the cache (external to the VLSI-BAM chip) which can be used by the cache controller logic to implement the "lock" and "unlock" multiprocessor synchronization operations [3]. Indexed load and store instructions (*ldx* and *stx*) are useful for the Prolog built-in predicate `arg/3` and for matrix operations in procedural languages.

Limited support is provided for detecting 32-bit two's complement overflow for addition and subtraction (*add32* and *sub32*). Addition by a constant (*addi*) is most often used for pointer arithmetic and so is available only as a non-trapping instruction.[5]The divide step (*divs*) and multiply step (*mpys*) instructions perform a single bit shift with conditional subtract/add. Thirty-two consecutive *divs* (*mpys*) instructions perform a complete 32-bit divide (multiply).

The special register file provides extra storage for system routines. Access to the special register file is accomplished using the *rd* and *wr* instructions.

Because the VLSI-BAM processor supports code which has branch instructions in branch delay slots, to correctly restart the pipeline after a trap, two PCs must be saved by a trap. The software trap instruction (*trap*) transfers control to a table of *jmp* instructions in low memory. Two instructions for each table entry are required for the *jmp* and its delay slot.

The remainder of this section motivates and presents our extensions to the general purpose instruction set. A major influence on the design of these extensions was the simultaneous development of an optimizing Prolog compiler. The abstract machine used by the compiler was initially designed using a top-down approach [31]. We assumed a set of data structures similar to those used by the WAM. Knowledge of possible compiler optimizations was applied to the semantics of Prolog to decompose Prolog's general operations into their components. These components, the abstract instruction set, are the instructions and addressing modes required to compile Prolog operations into efficient code. Efficient translation of abstract machine instructions into the architectural instruction set was a prime influence in the first pass of the instruction set design.

Touati and Despain [29] present numerous measurements of WAM execution behavior. Results of their studies were part of our starting point in the VLSI-BAM

[5] A trapped version of addition by a constant can be synthesized from *ldi* followed by *add32*.

Instruction	Operands	Action	Cycles
ld, ldl	r(i), disp16, r(k)	r(k) ← M[r(i)+disp16] *(ldl distinguishable to cache for synchronization operations)*	1
ldx	r(i), r(j), r(k)	r(k) ← M[r(i)+r(j)]	1
st, stu	r(i), r(k), disp16	M[r(k)+disp16] ← r(i) *(stu distinguishable to cache)*	1
stx	r(i), r(k), r(l)	M[r(k)+r(l)] ← r(i)	2
las	r(i), disp16, r(k)	r(k) ← M[r(i)+disp16]; M[r(i)+disp16] ← –1	2
add, sub, and, or, xor	r(i), r(j), r(k)	r(k) ← r(i) **op** r(j)	1
add32, sub32	r(i), r(j), r(k)	r(k) ← r(i) **op** r(j) *(trap on signed 32-bit overflow)*	1
addi, andi, ori, xori	r(i), imm16, r(k)	r(k) ← r(i) **op** imm16	1
sll, sra, srl	r(i), r(j), r(k)	r(k) ← r(i) **op** r(j)⟨4:0⟩	1
slli, srai, srli	r(i), imm5, r(k)	r(k) ← r(i) **op** imm5	1
divs, mpys	r(i), r(j), r(k)	(r(k), PQ, TF) ← **op**(r(i), r(j), PQ, TF)	1
cmp	cond, r(i), r(j)	TF ← (r(i) cond r(j))	1
bt	addr26	if (TF) PC⟨25:0⟩ ← addr26	1
btan	addr26	if (TF) PC⟨25:0⟩ ← addr26; else annul next instruction	1
btat	addr26	if (TF) { PC⟨25:0⟩ ← addr26; annul next instruction }	1
jmp	addr26	PC⟨25:0⟩ ← addr26	1
jmpr	r(i), disp16	PC ← r(i) + disp16	2
call	addr26	r(31) ← PC+1; PC⟨25:0⟩ ← addr26	1
rd	s(i), r(k)	r(k) ← s(i)	1
wr	r(i), s(k)	s(k) ← r(i)	1
trap	imm5	save PCs and PSW; set supervisor bit; PC ← 2*(32+imm5)	6
rft		restore saved PSW; fetch at saved PCs	4

TABLE 3.1. General Purpose Instructions

Tables 3.1–3.3 summarize the VLSI-BAM processor instruction set. The first two columns give the instruction mnemonic and operands. The third column gives the instruction's register transfer description. R(i) denotes general purpose register i; s(i) denotes special register i; disp$n$ is a sign-extended n-bit displacement; imm$n$ is a sign-extended n-bit immediate; addr26 is a 26-bit segment offset; off1_8 and off2_8 are zero-extended 8-bit displacements; tag is a four-bit immediate tag value; and cond is one of twenty comparison conditions. M[x] is the memory location at address x. Tag^value specifies the tag insertion operation. Tvar represents the value of the unbound variable tag (0001). Cycle counts assume no pipeline stalls due to load or store delays. All branch and jump instructions are delayed, and the following instruction is executed unless it is annulled. The cycle count of *dref* depends on the number of memory operations (l) performed.

design. For example, although we had included cdr-coding of lists in our earlier processor [9, 25], we found that it is not of sufficient benefit to justify hardware support. Therefore cdr-coding support was not provided in the VLSI-BAM. In the following subsections we provide additional WAM performance measurements that we found useful as a basis for making design decisions during the design of the VLSI-BAM processor.

In addition to our studies of abstract instruction sets, we investigated the microarchitectural requirements for high performance Prolog [5] and gathered execution statistics for the VLSI-PLM, a microcoded implementation of the WAM [25]. These investigations pointed out those microarchitectural features that would give the greatest performance gains and the Prolog operations that most need instruction set support.

3.1. Prolog Inspired General Purpose Instructions

Prolog inspired general purpose instructions are those instructions which support Prolog and which also may be useful in the implementation of other languages (Table 3.2). These instructions include load and store of immediates, single-cycle double-word load and store, and push and pop memory operations.

Immediates can be loaded, stored, or used in a comparison (*ldi*, *sti*, *stid*, *cmpi*). The immediates are tagged and are created by sign-extending a 12 or 17-bit immediate and replacing the four most significant bits with an immediate tag. Load immediate (*ldi*) is used for creating integers and atoms. Store immediate (*sti*) is an optimization of a *ldi, st* sequence and is used to bind an atom with a variable that is known at compile time to be unbound.

Double-word memory operations (*ldd, std, stdc, pushd, pushdc*) are motivated by Prolog's large memory bandwidth requirements. A double-word store or push is single-cycle (*stdc, pushdc*) only if the source registers form a consecutive, even/odd register pair, because only three registers, two of which must be adjacent, can be read from the register file per cycle. The *std* and *pushd* instructions allow the use of non-consecutive registers. They are two-cycle instructions, but this is offset by the absence of a pipeline stall when they are immediately followed by a memory operation.

Push instructions are included to support compound term creation. Using branch-and-bound search techniques, we determined an optimal set of single-cycle instructions for creation of all possible two and three-word structures [14]. This set of instructions is optimal in the sense that, for our microarchitecture, each structure is created in the smallest number of cycles. The resulting "compound term creation instruction set" favors the idiom of placing two words of data in registers and then moving them to memory using a double-word push. The VLSI-BAM chip also provides the external cache controller with a "push instruction" signal. With a properly designed external cache controller, push operations can skip the fill of the cache line from memory if a push incurs a cache miss and also refers to the first word of the cache line [6]. This optimization has been used in a previous Prolog design [20]. The push instructions allow the amount of the increment to be specified, and any general purpose register can be used as a stack pointer.

Prolog requires that variable assignment be undone on backtracking. This unbinding of variables is implemented by recording variable addresses on a "trail"

Instruction	Operands	Action	Cycles
ldi	tag, imm17, r(k)	r(k) ← tag^imm17	1
sti	tag, imm17, r(k)	M[r(k)] ← tag^imm17	1
stid	tag, imm12, r(k), disp5	M[r(k)+disp5] ← tag^imm12	1
cmpi	cond, r(i), tag, imm12	TF ← (r(i) cond tag^imm12)	1
ldd	r(i), disp11, r(k), r(l)	r(k) ← M[r(i)+disp11]; r(l) ← M[r(i)+disp11+1] *(r(i)+disp11 even)*	1
std	r(i), r(j), r(k), disp11	M[r(k)+disp11] ← r(i); M[r(k)+disp11+1] ← r(j) *(r(k)+disp11 even)*	2
stdc	r(i), r(k), disp16	M[r(k)+disp16] ← r(i); M[r(k)+disp16+1] ← r(i+1) *(i and r(k)+disp16 even)*	1
push	r(i), r(k), disp16	M[r(k)] ← r(i); r(k) ← r(k) + disp16	1
pusht	r(i), r(k), disp16	if (TF) { M[r(k)] ← r(i); r(k) ← r(k) + disp16 }	1
pushd	r(i), r(j), r(k), disp11	M[r(k)] ← r(i); M[r(k)+1] ← r(j); r(k) ← r(k) + disp11 *(r(k) even)*	2
pushdc	r(i), r(k), disp16	M[r(k)] ← r(i); M[r(k)+1] ← r(i+1); r(k) ← r(k) + disp16 *(i and r(k) even)*	1
pop	r(i), disp16, r(k)	r(k) ← M[r(i)−disp16]; r(i) ← r(i) − disp16	1
umin, umax	r(i), r(j), r(k)	r(k) ← unsigned_min/max(r(i), r(j))	1

TABLE 3.2. Prolog Inspired General Purpose Instructions

stack. The original WAM model requires two pointer comparisons to determine if trailing is necessary. Our implementation restricts variables to the global stack (which reduces the number of comparisons to one) and uses a compare instruction followed by a conditional push (*pusht*). The *pop* instruction is used during backtracking to retrieve variable addresses from the trail stack. The compiler can reduce the amount of trailing and detrailing through the use of flow analysis to determine when uninitialized variables [1] can be used (our use of uninitialized variables is different from [1]—we use the same tag for both initialized and uninitialized variables and determine at compile time when destructive assignment is safe).

The location of, and interaction between, the environment and choice point stacks is software defined. However, unsigned maximum (*umax*) is provided to simplify the management of the environment and choice point stack pointers when these stacks are intermixed. In this case, allocation occurs at the maximum of the two stack pointer values.

3.2. Prolog Specific Instruction Set Support

Prolog specific instructions are those instructions tailored specifically for efficient execution of Prolog (Table 3.3). These instructions support tagged pointer creation, two and three-way branch on tag, pointer dereferencing, and unification of atoms.

3.2.1. TAGGED DATA SUPPORT Pointer creation is accomplished by the load effective address (*lea*) instruction which calculates an address and then replaces the most significant four bits with an immediate tag. This instruction is used to create pointers to unbound variables and compound terms (lists and structures).

Type checking built-ins are supported with single-cycle compare-and-branch-on-tag instructions (*btgeq* and *btgne*). These instructions also allow the compiler to replace shallow backtracking with a conditional branch on an argument's tag.

Prolog allows unbound variables to be bound together. The resulting reference chain must be dereferenced before subsequent variable binding. WAM instructions always dereference their operands, often resulting in superfluous dereferencing. However, our optimizing compiler keeps track of which variables are dereferenced and generates explicit dereferences only when necessary. Implementing dereference as a single instruction reduces static code size and allows dereference memory reads to be pipelined, resulting in a tighter loop than the equivalent assembly code [11, 24]. We use the same tag value for both unbound variables and reference pointers (unbound variables are self referential). The dereference instruction (*dref*) is implemented as a sequence of internal opcodes. Because a single *dref* instruction could potentially require many execution cycles, it is interruptible and restartable.

All of the basic arithmetic and compare instructions (*add*, *sub*, *and*, *or*, *xor*, *cmp*) have a version that traps on 28-bit overflow. These instructions operate on the full 32-bit word, but 28-bit overflow occurs if either of the sources or the result do not have integer tags (0000 or 1111). The trap on 28-bit overflow allows Prolog arithmetic operations to be compiled to fast, safe code that avoids extra instructions for tag overflow checking. If a 28-bit overflow does occur, the trap routine can signal an overflow error or convert the data into an alternative representation.

3.2.2. UNIFICATION SUPPORT Unification is one of the primary operations of Prolog; it is used for argument passing, structure creation, structure decomposition, and pattern matching. Although general unification is a complex algorithm, if one is given information about the arguments being unified, the general algorithm can be greatly simplified. This is one of the advantages of the WAM instruction set over an interpreter. Our compiler takes this principle further and propagates information to simplify unification as much as possible.

Analysis of the primitives necessary to support unification of a Prolog variable with an atom [31] motivates the single-cycle unify-immediate instruction (*uni*) which binds the atom to the variable if the variable is unbound, and otherwise tests them for equality.

Unification of a Prolog variable with a compound term also benefits from special support. Analysis of the primitives necessary to support unification of a Prolog variable with a list or structure [31] motivates the switch-tag instruction (*swt*), a three-way branch based on the tag of one register. One direction of the branch is taken if the tag is an unbound variable; a second direction is taken if the tag

Instruction	Operands	Action	Cycles
lea	tag, r(i), disp12, r(k)	r(k) $\leftarrow$ tag^(r(i)+disp12)	1
btgeq, btgne	tag, r(i), disp16	if (r(i)$\langle$31:28$\rangle$ =/$\neq$ tag) { PC $\leftarrow$ PC + disp16; annul next instruction }	1
dref	r(i)	if (r(i)$\langle$31:28$\rangle$ = tvar) do { tmp $\leftarrow$ r(i); r(i) $\leftarrow$ M[r(i)] } until ((r(i)$\langle$31:28$\rangle$ $\neq$ tvar) or (r(i) = tmp)) *(l = number of memory refs)*	$l = 0$: 1 $l \neq 0$: $2 + 2l$
add28, sub28, and28, or28, xor28	r(i), r(j), r(k)	r(k) $\leftarrow$ r(i) **op** r(j) *(trap on non-integer tags)*	1
cmp28	cond, r(i), r(j)	TF $\leftarrow$ (r(i) cond r(j)) *(trap on non-integer tags)*	1
uni	tag, imm17, r(i)	if (r(i)$\langle$31:28$\rangle$ = tvar) { M[r(i)] $\leftarrow$ tag^imm17; TF $\leftarrow$ 0 } else if (r(i) = tag^imm17) TF $\leftarrow$ 0; else TF $\leftarrow$ 1	1
swb	r(i), r(j), off1_8, off2_8	if ((r(i)$\langle$31:28$\rangle$ = tvar) and (r(j)$\langle$31:28$\rangle$ $\neq$ tvar)) PC $\leftarrow$ PC + off1_8; else if ((r(i)$\langle$31:28$\rangle$ $\neq$ tvar) and (r(j)$\langle$31:28$\rangle$ = tvar)) { PC $\leftarrow$ PC + off2_8; annul next instruction } else annul next instruction	1
swt	r(i), tag1, tag2, off1_8, off2_8	if (r(i)$\langle$31:28$\rangle$ = tag1) PC $\leftarrow$ PC + off1_8; else if (r(i)$\langle$31:28$\rangle$ = tag2) { PC $\leftarrow$ PC + off2_8; annul next instruction } else annul next instruction *(tag1 or tag2 is tvar)*	1

TABLE 3.3. Prolog Instructions

Program	Argument Type (%)			Cost (cycles)	
get_list	variable	list	other	swt	two-way
prover	18.7	80.5	0.8	1.20	1.40
meta_qsort	42.1	42.0	16.0	1.58	2.32
simple_analyzer	24.4	67.4	8.3	1.33	1.74
chat_parser	8.8	84.8	6.4	1.15	1.37
average	23.5	68.7	7.9	1.32	1.71
get_structure	variable	structure	other	swt	two-way
prover	26.7	73.3	0.0	1.27	1.53
meta_qsort	37.6	62.4	0.0	1.38	1.75
simple_analyzer	13.5	86.5	0.0	1.14	1.27
chat_parser	44.0	52.5	3.5	1.48	1.98
average	30.4	68.7	0.9	1.31	1.64

TABLE 3.4. WAM Variable/Compound Term Unification Statistics

matches a specified immediate tag (usually list or structure); and a third direction is taken for all other tags. The three-way branch could be implemented using two two-way branches, however, WAM execution statistics (Table 3.4) show that there is a small but significant performance advantage to the three-way branch.

The LOW RISC processor [19] provides a 5-way branch and the Carmel-2 processor [11] provides a 10-way branch based on the tag of a single register. WAM execution statistics show that such generality is unnecessary for unification of a Prolog variable with a compound term.

When the compiler cannot determine any information about the types of the arguments to be unified, then general unification must be used. In this case one can still take advantage of dynamic properties of the argument types. The common cases of general unification should be done quickly in-line and infrequent cases passed to a general unification subroutine. Analysis of WAM execution (Table 3.5) indicates that about 70% of all general unifications are simple bindings of an unbound variable with a non-variable. These statistics motivate the switch-bind instruction (*swb*), a three-way branch based on the tags of two registers. The conditions of the three branch directions are: variable/non-variable, non-variable/variable, and otherwise (order of the arguments matters). This allows the common cases of variable/non-variable and non-variable/variable to be done in-line. A general unification subroutine is called for all other cases. Note that although the quick success and quick failure cases are simple to check for, their execution frequency is low enough that we have chosen not to do these checks in-line.

The Pegasus processor [24] supports general unification with a 16-way branch based on two tag bits from each of two registers. The LIBRA processor [18] has a "partial unify" instruction. This single-cycle instruction performs either a nop, a store, a call, or a branch depending on the tags and comparison of the two arguments. It executes the variable/non-variable case of general unification in four cycles (not counting dereferencing of the arguments). Using switch-bind (*swb*),

Program	Argument Type (%)					
	quick success	quick failure	var nonvar	nonvar var	var var	recursive
prover	15.6	15.6	0.0	61.4	0.0	7.5
meta_qsort	0.0	0.0	0.0	50.5	49.5	0.0
simple_analyzer	0.1	2.3	13.3	70.5	11.5	2.1
chat_parser	0.3	11.8	13.6	69.3	2.3	2.5
average	4.0	7.4	6.7	62.9	15.8	3.0

TABLE 3.5. WAM General Unification Statistics

VLSI-BAM executes this case in five cycles. Although the partial unify instruction of the LIBRA has a slight performance advantage, its complexity does not fit with our goal of minimally extending a general purpose architecture.

3.2.3. Example Use of Prolog Specific Instructions Figure 3.1 provides an example of the use of several of the Prolog specific instructions. The predicate (created just as an example) succeeds for only certain combinations of variables and atoms for the arguments. When the second argument is an atom, the arguments are unified. When compiling, we assume that mode analysis cannot deduce any information about the types of the arguments when the predicate is entered.

The VLSI-BAM code places the two predicate arguments in registers `r0` and `r1`. Before the type can be determined using a *swt* instruction, each argument must be fully dereferenced (using *dref*). The first *swt* instruction branches to label `example_2_1` when `r0` is an atom. Execution falls through when `r0` is neither an atom or an unbound variable. In this case, the fall-through corresponds to failure of the predicate, and so the `fail` routine is called. As an optimization, the delay slot of the `jmp(fail)` is filled with the first instruction of the `fail` routine and we now jump to the second instruction of `fail`.

The delay slot of the *swt* is executed when it branches to its first destination. Typically, the first instruction at the destination (`dref(r1)`) is replicated in the delay slot and the destination address incremented in order to reduce the execution time by a cycle.

When the two arguments are an unbound variable and an atom (in that order), then the unification reduces to the binding of a variable to a non-variable (`st(r1,r0)`). The variable being bound may require trailing, and this is done with a `cmp(ltu,r0,hb)`, `pusht(r0,tr,1)` sequence. When both arguments are atoms, the unification simplifies to an equality comparison (`cmp(ne,r0,r1)`, `btat(fail)`).

We will return to this example in Section 5 to help illustrate the methodology behind our cost-benefit analysis.

4. COMPILATION OF PROLOG

A significant aspect of our project was the simultaneous development of an optimizing Prolog compiler [31, 35]. The compiler incorporates techniques for determinism extraction and use of destructive assignment. The compiler accepts standard Prolog and produces code for a simple non-WAM abstract machine. Although the

```
example(X, Y) :- (var(X); atom(X)), atom(Y), X = Y.
example(X, Y) :- atom(X), var(Y).
```

```
procedure(example/2).
    dref(r0).
    swt(r0,tatm=example_2_1+1,tvar=example_2_2).
    dref(r1).
    jmp(fail+1).
    ldd(b-2,t0/t1).
label(example_2_1).
    dref(r1).
    swt(r1,tatm=example_2_3+1,tvar=example_2_4).
    cmp(ne,r0,r1).
    jmp(fail+1).
    ldd(b-2,t0/t1).
label(example_2_2).
    dref(r1).
    btgne(tatm,r1,example_2_0).
    st(r1,r0).
    cmp(ltu,r0,hb).
    jmpr(cp+1).
    pusht(r0,tr,1).
label(example_2_0).
    jmp(fail+1).
    ldd(b-2,t0/t1).
label(example_2_3).
    cmp(ne,r0,r1).
    btat(fail).
    jmpr(cp+1).
    nop.
label(example_2_4).
    jmpr(cp+1).
    nop.
```

FIGURE 3.1. Example Translation of Prolog into VLSI-BAM Instructions

compiler uses stacks and data structures similar to WAM implementations, it does not use the WAM during compilation, but instead directly compiles to its own abstract machine. Automatic mode generation (type inferencing) is implemented using abstract interpretation [8]. It derives ground, uninitialized variable [1], and dereference modes. Differences between the numbers listed in the following sections and those in a previous version of the paper [15] reflect improvements in the compiler and back-end instruction reorderer.

Compilation of Prolog is done in three stages. First, the compiler produces code for its abstract machine. Second, this code is macro-expanded into the VLSI-BAM instruction set. Finally, the VLSI-BAM code is optimized by a peephole optimizer and instruction reordering stage that maximizes the use of the double-word bus and minimizes the number of nops and pipeline stalls.

5. COST/BENEFIT ANALYSIS OF ARCHITECTURAL FEATURES AND INSTRUCTIONS

In Section 3 we motivated our instruction selection based on several sources of information: work on abstract instruction sets for compilers, bottom-up analysis of microarchitectural requirements for high performance Prolog, and analysis of WAM execution statistics. In this section we give a more rigorous validation of the architectural design and instruction selection by analyzing the cost and performance benefits of each special purpose feature and instruction. There has been some work to determine such results for other designs [10, 11, 24, 26], but the analysis presented here is more complete.

5.1. Cost of Features

Table 5.1 shows the implementation cost of those features that extend the VLSI-BAM beyond a general purpose architecture. Implementation cost is expressed in terms of chip area required to implement the feature and in terms of VLSI design effort required. The chip area is measured in percent of total active area which includes both transistor and wiring area. The chip contains approximately 110,000 transistors, and the total active area is 91 square millimeters using 1.2 μ CMOS (two metal layers). The VLSI layout was done using a symbolic layout editor with custom designed, parameterized cells. The building blocks were assembled into larger units using a data path compiler, PLA compiler, tiler, and router. The design effort for each feature is given as a percentage of its design that was automatically performed by the design tools. The last column of Table 5.1 lists those instructions that depend on a given feature. We do not give each feature's effect on the cycle time, since the microarchitecture and logic were designed carefully to prevent these features from being on the critical path.

Segment mapping requires the greatest area of the special features. This area is primarily due to the 32 by 24-bit register file which contains the segment map. This register file is used to extend the address space as well as perform tag mapping. A smaller register file tailored to tag mapping alone would take less area. The next greatest area consuming feature is the tagged-immediate generation circuitry. This is due in part to the use of three distinct instruction formats for tagged-immediates.

Feature	Active area	Design complexity	Instructions affected
segment mapping	4.8%	≈100% compiled	—
tagged-immediate	2.2%	100% compiled	ldi, cmpi, sti, stid, lea, uni
switch offset adders	2.0%	100% compiled	swt, swb
double-word memory port	1.9%	95% compiled; 5% by hand	ldd, std, stdc, pushd, pushdc, pop
fast tag logic	1.6%	≈100% compiled	btgeq, btgne, swt, swb, dref, uni
multi-cycle/conditional	0.1%	100% compiled	stx, std, pushd, pusht, dref, uni
tag overflow detect	≈0.0%	100% by hand (10 gates)	cmp28, add28, sub28, and28, or28, xor28
total special features	12.6%	99% compiled; 1% by hand	

TABLE 5.1. Cost of Special Architectural Features

For each special feature of the VLSI-BAM processor, this table gives the percentage of active area (transistors and wires) required to implement the feature, the design complexity of the layout, and a list of instructions that depend on the feature. The design complexity is given as a percentage of the layout that was automatically generated (using tilers, routers, etc.) and the percentage that was laid out by hand. ≈100% compiled indicates that less than 30 gates were placed by hand. Multi-cycle/conditional is a subset of internal opcodes—the 0.1% active area refers to the entire internal opcode implementation.

The three-way branch instructions, *swt* and *swb*, use a unique destination offset format and require two addition displacement adders to allow the destination address calculation to overlap with the opcode decode. The double-word memory port requires extra ports on the general purpose register file to support the increased bandwidth. The area listed is the difference in size between our four/five-port register file and the more usual three-port register file.[6]The extra pads required by the double-word bus are not included in the cost. After the fast tag logic, the remaining features use a very small portion of the total active area.

5.2. Benefits of Features

To determine the performance benefit of each feature, we calculated the cycle count increase caused by omitting the use of all instructions that depend on the feature [30]. For example, if omitting the instructions *ldd*, *std*, *stdc*, *pushd*, and *pushdc* increases execution time from 100 cycles to 111 cycles, then the performance benefit

[6]The *pop* instruction is included on the double-word memory port instruction list since it also requires the extra register file port.

`procedure(example/2).`	`procedure(example/2).`
	`label(expand_0).`
`    dref(r0).`	`    btgeq(tvar,r0,expand_2).`
	`label(expand_1).`
`    swt(r0,tatm=example_2_1+1,`	`    btgeq(tatm,r0,example_2_1).`
`        tvar=example_2_2).`	`    btgeq(tvar,r0,example_2_2).`
`    dref(r1).`	
`    jmp(fail+1).`	`    jmp(fail+1).`
`    ldd(b-2,t0/t1).`	`    ldd(b-2,t0/t1).`
	`label(expand_2).`
	`    ld(r0,r14).`
	`    cmp(eq,r0,r14).`
	`    btat(expand_1).`
	`    jmp(expand_0).`
	`    addi(r14,0,r0).`
`label(example_2_1).`	`label(example_2_1).`

FIGURE 5.1. Result of Removing *dref* and *swt* Instructions

due to the double-word memory port is 11%. An instruction is omitted by replacing it with its macro-expansion into instructions that still remain in the instruction set. An effort was made to determine optimal expansions, and after macro-expansion, peephole optimization and instruction reordering are performed. Omission of segment mapping requires that explicit instructions be inserted to mask tag bits before tagged-pointers are used as addresses. The combined benefit of two or more architectural features is determined by omitting all instructions affected by at least one of the features. A description of a preliminary version of this analysis is given in [23].

To illustrate the technique used to measure performance benefits, Figure 5.1 shows how the code from Figure 3.1 changes when the *dref* and *swt* instructions are removed from the instruction set. The work done by these two instructions must now be done using the *btg* instruction along with several general purpose instructions.

The *dref* instruction is replaced with a *btg* that jumps to an explicit dereference loop. The *dref* instruction takes 1 cycle when the initial tag is non-variable and 4 cycles when only one memory load is required for dereferencing. The explicit loop requires 1 cycle when the initial tag is non-variable and 7 or 9 cycles when one memory load is required. To gain performance (at the cost of code size), a separate dereference loop is generated for each *dref* instruction replaced. Use of a single dereference subroutine would require saving the current PC and would take extra execution cycles.

The `swt(reg,tag1=label_1,tag2=label_2)` instruction requires 1, 2, and 2 cycles for the `label_1`, `label_2`, and fall-through branch directions. The *btg* instruction pair requires 2, 3, and 2 cycles, which is, on average, almost one cycle more for each dynamic occurrence.

Table 5.2 lists the performance benefit of the features given in Table 5.1. Fast

Benchmark	Feature Performance Benefit (%)						
	fast tag logic	double-word memory port	segment mapping	multi-cycle conditional	tagged-immediate	tag overflow detect	all Prolog support
boyer	16.1	6.4	13.9	11.0	1.3	5.6	64.9
chat_parser	12.2	18.6	10.8	9.6	8.8	0.0	66.1
meta_qsort	15.2	18.6	13.2	11.9	9.4	0.6	71.8
nand	9.0	18.0	5.8	4.1	3.7	2.8	58.5
peep	8.3	13.2	6.0	5.1	4.7	4.9	57.8
prover	12.5	17.6	9.2	7.7	8.7	0.0	67.6
simple_analyzer	13.5	9.6	13.5	11.6	4.7	4.9	62.7
average	12.4	14.6	10.4	8.7	5.9	2.7	64.2

TABLE 5.2. Performance Benefit of Special Architectural Features

This table gives the percent performance benefit for each special feature of the VLSI-BAM processor. The last column of lists the performance benefit of segment mapping and all instructions given in Tables 3.2 and 3.3. All benchmarks are compiled with automatic mode generation, and cache effects are not included.

tag logic, double-word memory port, segment mapping, multi-cycle support, and tagged-immediate support are consistently important features. Tag overflow detection is important only in programs that make heavy use of integer arithmetic. The overall Prolog support column is determined by using only the instructions from Table 3.1 (and non-tagged versions of *ldi* and *cmpi*), omitting segment mapping and all instructions in Tables 3.2 and 3.3.

Each of the benefits listed in Table 5.2 represent performance changes with respect to the full VLSI-BAM architecture and instruction set. In practice, however, the benefit of a feature depends on what other features are also present. To study the interaction of the architectural features, the performance of every combination of features was determined.

Figure 5.2 is a plot of the cost versus benefit of all 48 meaningful combinations of the special architectural features (except tag overflow detection—each of these 48 points assumes that tag overflow detection is supported). The upper left hand portion of the plot contains two additional points that represent the base case and the base case plus the instructions *push*, *umin*, and *umax*. The line is the lower half of the convex hull of the points and represents one strategy for adding features to the general purpose base. Each line segment connects the points so that the best benefit per cost is achieved as one moves from left to right. In this case, it turns out that each point on the line represents a set of features that adds one new feature to that of the point on its left.

Starting from the base architecture, we first add the three instructions (*push*, *umin*, *umax*) that require no special architectural support, but which were not included in our base instruction set. Then tag overflow hardware is the next best feature to add, since the amount of hardware is negligible. It is interesting to note that the SPARC architecture also added tag overflow support as its one addition for supporting tagged languages. The remaining features, moving left to right, are

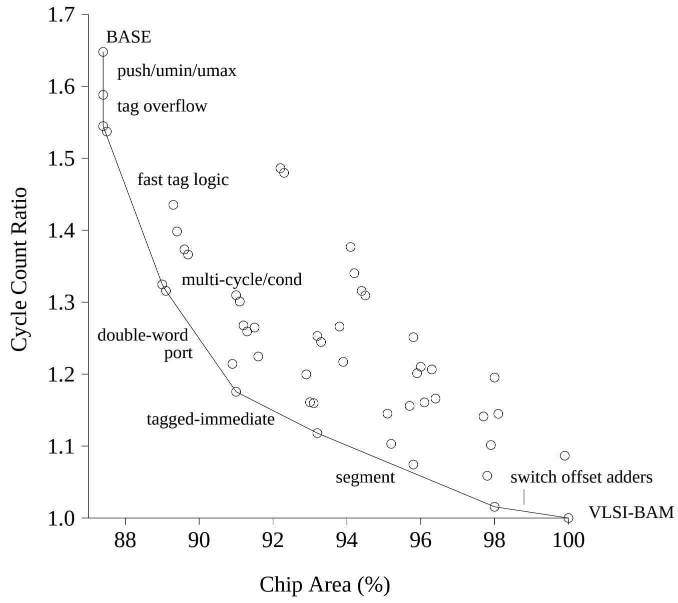

FIGURE 5.2. Benefit *vs.* Cost of Architectural Features

fast tag logic, multi-cycle and conditional instructions, double-word memory port, tagged immediates, segment mapping, and finally switch offset adders.

To summarize, the specialized support added for Prolog does not require unreasonable amounts of chip space or hand layout (13% active area for all Prolog related features), and it provides a performance benefit of 60 to 70%.

5.2.1. Comparison with Korsloot and Mulder Korsloot and Mulder [17] present a study of the benefits of special architectural support for Prolog. They conclude that support for tags, auto increment/decrement addressing, and special choice point buffers can provide a 20 to 25% performance advantage. This is a smaller advantage than given in Table 5.2 (64% for all Prolog support), so it is important to investigate the source of the discrepancy. In this subsection we modify the VLSI-BAM benefit analysis to match as closely as possible the Korsloot and Mulder study. The primary source of the discrepancy is due to tag placement. A secondary contribution to the discrepancy arises from the benefit of specialized VLSI-BAM instructions that are not considered in the Korsloot and Mulder study.

One of the main differences between the assumptions of the two studies is the placement of the tag bits in the 32-bit word. The VLSI-BAM places the tags in most significant bits of the word, whereas Korsloot and Mulder assume tags in the least significant bits. Korsloot and Mulder break tag support down into three categories: tag masking (to use tagged data as a memory address), tag insertion (to create tagged data), and tag extraction (primarily to branch based on the tag value). When the tag bits are in the least significant bits, tag masking can be done using displacement addressing. Tag insertion is simply addition of a small constant (for immediates, this can be done at compile time).

Benchmark	Cycle Count Ratio		
	base	tag support (masking)	tag support (insertion)
boyer	1.000	0.963	0.897
chat_parser	1.000	0.961	0.894
meta_qsort	1.000	0.955	0.880
nand	1.000	0.971	0.897
peep	1.000	0.974	0.893
prover	1.000	0.959	0.879
simple_analyzer	1.000	0.964	0.891
average	1.000	0.964	0.890
KM boyer	1.00	0.98	0.98
KM chat_parser	1.00	0.98	0.98

TABLE 5.3. Tag Support Comparison with Korsloot and Mulder

This table compares our architectural studies with the results of Korsloot and Mulder [17]. The last two rows of data are from [17]. The top rows are data collected with our tools. The column labeled "base" represents the VLSI-BAM general purpose base. The following instruction additions to this base are made for each column: "tag support (masking)", segment mapping; and "tag support (insertion)", *ldi*, *cmpi*, and *lea*.

These differences result in a different conclusion about the performance benefits for tag masking and tag insertion hardware support. Table 5.3 gives the cycle count reduction resulting from architectural support for tag masking and tag insertion. Both of these results are relative to our base architectures. The results show a significant difference in the benefit of hardware support for tag masking and tag insertion. The benefit is small when the tag bits are in the least significant bits (2% for masking and 2% for insertion), but more sizable when the tags are in the most significant bits (4% and 12%, respectively). Tag placement in the least significant bits shows less benefit due to specialized architectural support, therefore this tag placement has better performance when there is no specialized support.

When there is no architectural support the essential advantage of placing the tag in the lower bits is that tag insertion and tag masking can be done as the addition of a small constant that can be combined with immediates and displacement constants. When the tag is in the most significant bits of the word, then combining the constant for the tag operation with the offset requires an immediate with bit fields specified at both ends of the word—not an immediate format that is usually supported.

Overflow detection during tagged arithmetic is simpler with the tags in the low bits, because the value of the tagged integer lies in the upper part of the word, and an overflow of this value is detected by the overflow hardware already present for 32-bit arithmetic. This advantage is not of the greatest importance, since the time required for overflow detection even in the case of most-significant bit tags with no hardware support is modest (2.6% from Table 5.2).

The proper choice for tag placement appears to depend strongly on whether architectural support will be provided for tag operations. Obviously if a Prolog system is to be implemented on hardware that includes no support for tags, then there are compelling reasons to place the bits in the least significant bits. But once one is willing to commit hardware for tag support, the choice becomes more difficult. Tags in the most significant bits can be of uniform width, simplifying the branch on tag support. For tags in the least significant bits the advantage of combining tag constants with immediates and offsets becomes less significant because formats for immediates can be changed to include bits from both the high and low ends of the word. Also, the amount of hardware required to support tagged-arithmetic overflow detection is minimal in either case.

Returning to the comparison of our analysis with that of Korsloot and Mulder, we feel that the best way to compare architectural support, other than for tag masking and tag insertion, is to assume that our base architectures already include support for tag masking and tag insertion. Although this is not exactly what the results of Korsloot represent, their numbers should be very close since the incremental effect of tag masking and tag insertion is only 1%.[7]Specifically, the base architecture that we assume consists of the VLSI-BAM base plus segment mapping and instructions needed to support tag masking and insertion. This configuration is labeled "KM base" in Table 5.4.[8] Using this base, we present results in Table 5.4 for the advantages of adding architectural features to support tag extraction, address modification, and choice point support.

For tag extraction Korsloot adds a branch on tag instruction. This is comparable to the *btgeq* and *btgne* instructions in the VLSI-BAM. Adding *btg* instructions to the KM base results in similar cycle count reduction as observed by Korsloot and Mulder.

Address modification consists of adding auto increment and auto decrement addressing modes. This is the same as adding the VLSI-BAM push and pop instructions (excluding the double-word versions). In this case, we observed less reduction in cycle count, apparently because our instruction reorderer is able to combine multiple *addi* instructions arising from macro expanding *push* instructions (*push* is replaced with *st* and *addi*). Specifically, for boyer and chat_parser, the increase in dynamic *addi* count (resulting from disallowing auto address mode) is 3.3 times less that the dynamic count of *push* and *pop*. This ratio roughly matches the ratio of the reductions in cycle count: $(1 - 0.96)/(1 - 0.989) = 3.6$ for boyer.

Korsloot and Mulder provide choice point support by adding a shadow buffer that allows quick transfer of between the buffer and the register file. The VLSI-BAM design supports choice points by relying on compiler optimization and the use of double-word loads and stores. The "choice point support" column in Table 5.4 shows that double-word loads and stores provide a slightly better performance improvement (3%) over a specialized choice point buffer. This is because double-word memory operations can also be used for creating procedure environments and accessing adjacent arguments of lists and structures.

When the support for tags, address modification, and choice points are all com-

[7]The benefit of support for tag masking, insertion, and extraction is just 1% more than the benefit of support for only tag extraction.

[8]We also include tagged arithmetic overflow support in the KM base, because it was not included in the Korsloot study.

Benchmark	Cycle Count Ratio					
	KM base	tag support (extraction)	address modification	choice point support	SRISC	VLSI-BAM
boyer	1.000	0.912	0.989	0.958	0.853	0.735
chat_parser	1.000	0.945	0.972	0.879	0.792	0.704
meta_qsort	1.000	0.929	0.983	0.888	0.791	0.701
nand	1.000	0.916	0.985	0.876	0.772	0.738
peep	1.000	0.924	0.986	0.904	0.813	0.759
prover	1.000	0.910	0.983	0.885	0.769	0.708
simple_analyzer	1.000	0.928	0.994	0.938	0.852	0.747
average	1.000	0.923	0.985	0.904	0.805	0.727
KM boyer	1.00	0.90	0.96	0.94	0.81	—
KM chat_parser	1.00	0.94	0.90	0.93	0.76	—

TABLE 5.4. Comparison with Korsloot and Mulder

This table compares our architectural studies with the results of Korsloot and Mulder [17]. The last two rows of data are from [17]. The top rows are data collected with our tools. The column labeled "KM base" represents the VLSI-BAM general purpose base plus segment mapping, *ldi*, *cmpi*, *lea*, and 28-bit arithmetic. The following instruction additions to this base are made for each column: "tag support", *btgeq* and *btgne*; "address modification", *push* and *pop*; "choice point support", *ldd*, *std*, and *stdc*; "SRISC", *btgeq*, *btgne*, *push*, *pop*, *ldd*, *std*, *stdc*, *pushd*, and *pushdc*; and "VLSI-BAM", SRISC plus all the remaining VLSI-BAM instructions.

bined (the "SRISC" column), the total benefit is a cycle count reduction by 14 to 23%. For boyer and chat_parser, we find a slightly smaller reduction (3 to 5%) than Korsloot and Mulder.

The additional VLSI-BAM instructions beyond "SRISC" (*ldx*, *sti*, *stid*, *pusht*, *umin*, *umax*, *dref*, *uni*, *swb*, and *swt*) provide an additional 3 to 10% performance advantage. Section 5.3 provides more detail on which instructions contribute the most to this improvement.

Note that the total benefits for special purpose Prolog support found here average to 37.6% (1/0.727 = 1.376) compared to 64.2% (Table 5.2). This difference, as we discussed earlier, is due to the effects of tag placement on the apparent architectural advantage of tag masking and tag insertion.[9]

Two primary conclusions can be drawn from the results of the comparison with Korsloot and Mulder. First, if there is no architectural support for tag operations, placing the tags in the least significant bits gives better performance. With architectural support, however, there is no performance advantage to either tag position. The choice is primarily dictated by aesthetics or historical preference. Second, use of double-word load and store instructions gives as much or more performance improvement than specialized choice point buffers.

[9] $1.642/1.376 = 1.20 = 1.04 \times 1.12 \times 1.03$ where the three factors are for tag masking, tag insertion, and tag overflow detection.

5.2.2. Effect of Compiler Optimizations Andrew Taylor has reported performance results for an optimizing Prolog compiler (PARMA) targeted for the MIPS processor [28]. His work raises the question of whether specialized hardware can be completely replaced with compiler optimization. The combination of PARMA and the MIPS R3000 is roughly equivalent in speed with the combination of Aquarius and the VLSI-BAM (after adjusting for differences in clock rates). This implies that the PARMA compiler is able to find more performance enhancing optimizations than the Aquarius compiler. But special purpose architectural features could also speed up the PARMA/MIPS combination. Most likely, however, the performance advantages of special features would be reduced.

In this section we attempt to estimate the usefulness of our special architectural features given an improved compiler. A version of the PARMA compiler which generates VLSI-BAM code is not available, so to approximate the quality of code given by such a compiler, we modify the assembly code produced by the Aquarius compiler based on an analysis of the execution trace of each benchmark. This analysis determines which operations, at least for the specific inputs used, are not productive for achieving the solution. Because the analysis is based only on a limited set of the possible inputs, the resulting optimizations are "unsafe" as far as producing correct code is concerned, but the analysis allows us to approximate the code produced by an extremely good compiler.

Analysis of the trace determines which operations are not taking one closer to the solution. For example, if a particular branch is never taken, then that branch can be removed. If the new value written to a register is always identical to the old value it replaces, then that register update is unnecessary (the same holds true for memory locations). A specific case of this is when a dereference instruction does not result in a new value being loaded into the register—such a dereference instruction would be removed by our hypothetical "optimizing compiler." Also, register and memory liveness can be computed from the trace, and if a memory location (or register) is not read between the time of two writes, then the first write is unnecessary.

Unfortunately, our analysis does not approximate some important optimizations which, if implemented, could possibly have a large effect on the performance benefit of specialized architectural features. For example, we do not take into account the effects of global register allocation, determinism extraction, multiple specialization, and compile-time garbage collection [32]. Consequently, our results do not bound the improvements possible with compiler optimizations.

We performed the following steps to create optimized code. First, the program was executed and a trace of procedure calls was obtained. This trace contains information on the types of each of the arguments. Using the data types of the arguments, mode declarations were added to the program to supplement the compiler's static analysis. Then an instrumented instruction-level simulator was used to gather information on register (and memory) writes that did not change the value of the write's destination. Instructions that never place a new value in the destination of the write were eliminated. Next, all branch instructions that use the fast tag logic (*btgeq*, *btgne*, *swt*, and *swb*) were analyzed to determine if one direction was always taken. If the branch is always taken, it was replaced with a *jmp*. If the branch is never taken, it is removed. Also, register (and memory location) liveness was determined. If an instruction always writes a value that is never used, the instruction is removed.

Benchmark	Feature Performance Benefit (%)						
	fast tag logic	double-word memory port	segment mapping	multi-cycle conditional	tagged-immediate	tag overflow detect	all Prolog support
meta_qsort	4.0	20.9	8.3	8.4	11.2	0.8	57.1
nand	2.5	20.4	5.9	4.0	3.1	3.3	48.8

TABLE 5.5. Performance Benefit of Special Architectural Features Assuming an Improved Compiler

This table gives the percent performance benefit for each special feature of the VLSI-BAM processor after optimization of the compiled code to simulate the effect of an improved compiler. The last column of lists the performance benefit of segment mapping and all instructions given in Tables 3.2 and 3.3. Cache effects are not included.

Table 5.5 lists the performance benefit of the architectural features assuming an improved compiler. This table should be compared with Table 5.2. We see that the fast tag logic becomes much less important (it drops from 15% to about 3% performance benefit). This arises mostly from the near elimination of the dereference operation (see Tables 5.7 and 5.8). All of the other architectural features have roughly the same benefit as before. Thus, the primary architectural needs for Prolog, using the improved compiler, are high memory bandwidth (multi-word loads and stores) and fast tag masking and tag insertion. For the VLSI-BAM these tag operations are done using segment mapping and tagged-immediate formats. If the architecture is byte-addressed and the implementation places the tags in the least-significant bits, then these tag operations can be performed using displacement addressing. The need for multi-word memory loads and stores, however, remains.

Table 5.6 shows that specialized hardware support is still significant (31% for nand and 34% for meta) with the improved compiler. In Section 5.2.1 we saw that the KM base roughly corresponds to a general purpose base with a Prolog implementation that places tags in the lower bits. With the Aquarius compiler, specialized hardware of the VLSI-BAM provides slightly more speed up (36 and 43%, respectively). The reason that the speed up due to specialized hardware changes only slightly as compiler optimizations improve is that in spite of the extra optimizations, Prolog still requires significant bandwidth to and from memory. Also, branch-on-tag support will always be useful due to the imperfect knowledge at compile time (for example, we do not know the data types of some inputs until run-time).

The performance benefit of the improved compiler over the Aquarius compiler is 21 and 26% (Table 5.6). Using data from Taylor's thesis [28], PARMA's improvement over Aquarius could be as much as 40% for nand. The extra optimization capabilities of PARMA over our hypothetical compiler points to the need for additional benefit studies using a compiler as good as or better than PARMA.

Benchmark	KM base	Compiler optimizations
meta_qsort	33.8	25.9
nand	31.1	21.4

TABLE 5.6. Architectural Benefits *vs.* Compiler Benefits

This table compares the performance benefits of architectural features and those due to additional compiler optimization. The column labeled "KM base" is the performance difference between the KM base instruction set (Table 5.4) and the VLSI-BAM instruction set, both using the "improved" compiler optimizations. The "Compiler optimizations" column gives the performance difference of the "improved" compiler and the Aquarius compiler with static mode analysis, both using the VLSI-BAM instruction set. Cache effects are not included.

5.3. Benefits of Individual Instructions

Table 5.7 lists performance benefits of individual instructions or instruction groups. Significant (greater than one percent) performance benefit is obtained from a majority of the special purpose instructions (*dref, umin/umax, btgeq/ne, push/d/c, lea*, and *swt*). The multi-cycle pointer dereference instruction (*dref*) has an average execution time of 1.6 cycles. Macro-expansion of *dref* into an explicit loop increases the average dereference time to 2.2 cycles. Although the benefit of *dref* per dereference is only 0.6 cycle, the total performance benefit is significant because of its frequent use. Some of the smaller benchmarks (not listed in the table), however, show no benefit for *dref* due to the complete elimination of dereferencing by compiler optimization. Unsigned maximum (*umax*) is used during environment and choice point creation. Omission of *umax* causes the time to determine the top of stack to increase from one to three cycles. Tagged-pointer creation (*lea*) is a frequent operation, and its omission adds an extra cycle for tag insertion (using *or*). Elimination of auto-increment addressing (*push, pushd, pushdc*) requires one extra cycle for each block allocation. The three-way branch on tag (*swt*) can be replaced by two *btgeq* instructions, adding an extra cycle to two of the branch directions. Elimination of the two-way branch on tag (*btgeq/ne*) would require a two instruction compare and branch.

Compared with our previously published results [15], the performance benefit of *swt* has dropped significantly. The difference is due to improvements in the compiler and instruction reorderer that allow *btg* instructions to replace *swt* in special cases.

The remaining instructions have less than one percent average performance benefit. Because the VLSI-PLM spends about 5% of its time trailing variable addresses, we included special support in the VLSI-BAM (*pusht*). However, due to the compiler's use of uninitialized variables, which do not have to be trailed, trailing time is reduced to 1.4% in the VLSI-BAM. Omitting *pusht* causes a slow down of 0.6%, which corresponds to trail time increasing from 2 to 3 cycles. Preliminary analysis using macro-expanded WAM for the chat_parser benchmark indicated that the benefit for *pop* would be 1.5%. Compiler optimization of trailing has reduced this

Benchmark	Instruction Performance Benefit (%)										
		umin	btgeq	push						sti	
	dref	umax	btgne	pushd/c	lea	swt	pusht	swb	pop	stid	uni
boyer	8.1	4.5	2.6	2.0	1.2	1.7	0.8	1.1	0.0	0.0	0.0
chat_parser	3.5	3.6	3.5	2.5	3.0	0.9	1.3	0.6	0.8	0.3	0.9
meta_qsort	5.3	5.4	5.7	3.2	3.2	0.5	0.8	0.5	0.2	0.2	0.1
nand	1.0	3.2	3.1	3.0	1.8	0.2	0.1	0.0	0.0	0.0	0.0
peep	1.9	3.2	2.6	2.0	2.0	1.1	0.3	0.7	0.1	0.1	0.0
prover	2.8	3.6	3.8	3.2	2.7	1.0	0.4	0.4	0.2	0.4	0.0
simple_analyzer	7.9	3.6	1.6	1.9	2.0	1.8	0.3	0.4	0.0	0.1	0.0
average	4.3	3.9	3.2	2.5	2.3	1.0	0.6	0.5	0.2	0.2	0.1

TABLE 5.7. Performance Benefit of Individual Instructions

This table gives the percent performance benefit for each special instruction of the VLSI-BAM processor. All benchmarks are compiled with automatic mode generation, and cache effects are not included.

result. Similarly, compiler optimization reduces the number of general unifications, minimizing the benefit of *swb*. Our initial studies also overestimated the benefits of special support for unification of atoms (*uni*, *sti*, *stid*). Every time one of the atom unification instructions is executed, one cycle is gained when compared to not having the instruction. However, they simply do not have a high enough dynamic execution frequency to make a significant impact on the performance. Although *pusht*, *swb*, *pop*, *uni*, *sti*, and *stid* provide marginal performance benefit, their implementation uses only features already required by other instructions.

An interesting conclusion about the number of directions needed in multi-way branches can be made from these measurements. Multi-way branches are implemented in the VLSI-BAM with the *swt* and *swb* instructions, which are both single-cycle three-way branches (Table 3.3). *Swt* is used for unification of compound terms, for which greater than a three-way branch is not needed (Table 3.4 and [31]). *Swb* is used for unification of terms whose types are unknown at compile time. It takes care of 70% of these cases (Table 3.5), which gives a 0.5% execution time improvement (Table 5.7). If some single-cycle branch took care of 100% of these cases, we calculate the further improvement would be about 0.2%. Given the additional complexity that such a branch implies, we conclude that a multi-way branch with more than three directions is not effective for Prolog.

Table 5.8 gives the performance benefit of the special purpose instructions assuming an improved compiler. As mentioned before, the benefit of the *dref* instruction is drastically reduced because most dereference instructions do no useful work and are eliminated. The benefit of the *btgeq* and *btgne* instructions is reduced approximately in half, since a sizable number of these instructions always take the same branch direction. All of the remaining instructions remain at about the same level of benefit. Thus, a set of push instructions remains important along with an unsigned maximum for stack pointer manipulation.

Benchmark	Instruction Performance Benefit (%)										
	dref	umin umax	btgeq btgne	push pushd/c	lea	swt	pusht	swb	pop	sti stid	uni
meta_qsort	0.6	5.3	3.0	3.6	4.0	0.4	0.6	0.0	0.2	0.2	0.1
nand	0.3	3.7	2.2	3.4	2.2	0.1	0.1	0.0	0.0	0.0	0.0

TABLE 5.8. Performance Benefit of Individual Instructions Assuming an Improved Compiler

This table gives the percent performance benefit for each special instruction of the VLSI-BAM processor assuming improved compilation. Cache effects are not included.

	KCM / PLM	VLSI-BAM / PLM	SPARC / PLM	SPUR / PLM
bytes	3.0	3.1	6.5	14.1
instructions	1.1	2.6	5.5	12.0

TABLE 5.9. Static Code Size Ratios

5.4. Code Size Benefits

Table 5.9 compares the static code sizes of the VLSI-BAM, the KCM [2], Aquarius Prolog (SPARC) [34], and SPUR [4] relative to the PLM, a microcoded implementation of the WAM [9]. The VLSI-BAM and Aquarius SPARC code size is calculated using prover, meta_qsort, simple_analyzer, and chat_parser. The KCM code size is from [2]. The SPUR code size is from [4]. The sizes do not include symbol table or run-time library.

Macro expansion of WAM code into SPUR instructions causes the large code size of the SPUR. Static code size for the VLSI-BAM is surprisingly small, only slightly larger than that of the KCM. Measurements of the code size resulting when just using the VLSI-BAM's general purpose base instructions show that without the special architectural features for Prolog, the code size would be 2.1 times that of the VLSI-BAM. This is exactly the ratio observed between VLSI-BAM code and Aquarius Prolog SPARC code. Thus, the compactness of the VLSI-BAM code is due to the success of flow analysis in reducing code size (overcoming simple, but verbose macro-expansion) and the appropriateness of the VLSI-BAM instruction set for Prolog.

Table 5.10 gives the reduction in code size provided by the addition of each instruction (or instruction group). Past studies [4] have shown that code expansion of Prolog can be especially serious for simple instruction sets. One possible rule of thumb in cases where code size is important is that an instruction added for code size reduction should provide at least a 10% benefit. Using this criterion, only *dref* qualifies. One can also consider the combined benefit in terms of both performance

Benchmark	Instruction Code Size Benefit (%)										
	dref	umin umax	btgeq btgne	push pushd/c	lea	swt	pusht	swb	pop	sti stid	uni
boyer	17.8	2.2	1.9	4.4	12.9	−0.1	3.8	0.9	0.0	0.1	1.9
chat_parser	28.6	3.8	4.5	3.5	4.6	−0.2	8.0	0.3	0.0	2.9	10.5
meta_qsort	20.9	3.8	2.7	4.5	13.5	0.0	2.6	0.5	0.0	0.3	1.3
nand	23.3	3.8	1.0	3.8	8.6	−0.1	0.7	0.0	0.0	1.1	0.0
peep	36.8	2.1	2.8	4.4	9.0	−0.4	5.8	1.6	0.0	0.3	2.6
prover	17.0	3.2	0.9	4.6	9.7	0.0	2.1	0.6	0.0	0.6	0.0
simple_analyzer	25.6	3.4	1.8	3.9	9.4	0.0	3.5	0.5	0.0	0.8	2.8
average	24.1	3.2	2.2	4.2	9.6	−0.1	3.8	0.6	0.0	0.9	2.7

TABLE 5.10. Static Code Size Benefit of Individual Instructions

This table gives the percent static code size benefit for each instruction of the VLSI-BAM processor. All benchmarks are compiled with automatic mode generation, and the Prolog built-in procedures are not included in the size.

and code size and in this case *pusht* can be added to the list of justified instructions.

The *swt* shows a slight negative code size benefit because its destination offset field often overflows. For example, if the destination `label_2` is too far away, then

```
    swt(r1,tvar=label_1,tstr=label_2).
    <delay slot instruction>
```

is replaced by

```
    swt(r1,tvar=label_1,tstr=skip).
    <delay slot instruction>
    ...
skip:
    jmp(label_2).
    <another delay slot instruction>
```

But when *swt* is eliminated from the instruction set, it is replaced by two consecutive *btg* instructions (see Figure 5.1). The *btg* instructions have a much greater offset range, and so are not likely to overflow and need replacement. A *swt* instruction that overflows for both destinations requires more code space than just two *btg* instructions.

5.5. Summarizing the Costs and Benefits

In this section we have looked at special purpose support for Prolog from several perspectives. Here we summarize our findings by giving our opinion on what hardware support is best for Prolog, taking into account trends toward further compiler improvement.

If we assume that the implementation places the tags in the lower bits of the word, then hardware support for tag masking and tag insertion is supplied by base plus displacement addressing and normal load immediate and add immediate instructions.

The greatest need, given both current and future compilers, is adequate bandwidth to and from memory. This can be provided by multiple-word loads and stores. Fortunately, multi-word memory loads and stores are now part of most high-performance microprocessor instruction sets.

The most important specialized support for Prolog can be reduced to a fast (single cycle) branch on tag (equality or non-equality comparison with an immediate value). When the tags are placed in the low bits of the word, this usually forces the tags to be variable size. Unfortunately, this complicates a special branch on tag instruction. One solution is to support only the most common (or the two most common) tag placement and width. Less common tag formats must be checked using a multi-instruction sequence of a mask operation followed by a normal compare and branch.

The VLSI-BAM supports three-way branch-on-tag instructions, but the indications are that they are currently at the margin of usefulness (*swt* provides only a 1% performance advantage and *swb* is less), and future compilers will further reduce this usefulness. Also against three-way branches is the need for additional displacement adders and the restricted range of branch offsets.

The single most useful specialized instruction in the VLSI-BAM instruction set is *dref*. Not only does it provide a 4% performance benefit, but also a 24% reduction in static code size. However, trace analysis shows that most *dref* instructions could possibly be removed by a better compiler. Whether a dereference instruction should be included in future Prolog instruction sets is an open question, and depends on the effort of compilation that goes into the majority of code executed. To achieve a more interactive system, one may not spend much time on static analysis, and so a dereference instruction would be useful for reducing code size.

Although the performance benefit is small, the cost of supporting a tag check on tagged arithmetic operations is so small (a few gates) that such support should be considered. It is not surprising, then, that the SPARC instruction set supports tagged add and subtract instructions.

There are two other instructions that improve Prolog performance that can be considered "general purpose," but that are not present in many of the high-performance microprocessor instruction sets. Auto-increment addressing mode for stores (push instructions) speeds up stack operations and heap data creation. Management of interleaved stacks can be done more efficiently with the unsigned maximum operation. Both push and unsigned maximum maintain their performance enhancement as compiler optimization improves. The utility of the unsigned maximum instruction, however, remains to be verified for the case when the environment and choice point stacks are not interleaved.

In summary, future high-performance instruction sets for Prolog should include the following additions to a general purpose base: multi-word memory loads and stores, a single cycle branch on tag, tagged arithmetic support, push instructions, and (possibly) unsigned maximum.

6. PERFORMANCE RESULTS

Benchmark	KCM		Aquarius SPARC		Aquarius VLSI-BAM	
log10	0.039	(1.75)	—		0.0223	(1.00)
ops8	0.059	(2.09)	—		0.0282	(1.00)
times10	0.082	(2.05)	—		0.0400	(1.00)
divide10	0.091	(2.02)	—		0.0450	(1.00)
nreverse	0.65	(3.22)	0.440	(2.18)	0.202	(1.00)
qsort	1.32	(4.49)	0.520	(1.77)	0.294	(1.00)
serialise	1.22	(1.73)	1.64	(2.33)	0.704	(1.00)
query	12.6	(2.20)	15.1	(2.64)	5.73	(1.00)
mu	—		2.86	(2.36)	1.21	(1.00)
prover	—		3.20	(2.32)	1.38	(1.00)
queens_8	—		3.20	(1.88)	1.70	(1.00)
meta_qsort	—		17.2	(2.44)	7.06	(1.00)
nand	—		57.0	(2.32)	24.6	(1.00)
simple_analyzer	—		85.5	(1.71)	50.1	(1.00)
poly_10	—		112	(2.11)	53.2	(1.00)
chat_parser	—		550	(2.75)	200	(1.00)
boyer	—		3100	(1.45)	2140	(1.00)
geometric mean		(2.32)		(2.14)		(1.00)

TABLE 6.1. Performance Results

This table compares the performance of VLSI-BAM with that of two other Prolog implementations—the KCM and Aquarius Prolog running on a SPARCstation 1+ (25 MHz). Each result is presented as a time in milliseconds followed in parentheses by the ratio to the VLSI-BAM time. The KCM results [2] are derived from actual measurements of a system running at 12 MHz. The VLSI-BAM results are simulated assuming a 20 MHz clock and 128 KB instruction and data caches [6]. Results involving the Aquarius compiler (both SPARC and VLSI-BAM) use automatic mode generation. The programs used in this table and elsewhere in this paper include the well-known Warren benchmarks (the first eight in the table), of which query is modified to use integer division in place of the original floating point; mu, which proves a theorem of Hofstadter's "mu-math"; prover, a simple theorem prover; queens_8, which solves the eight queens problem using an incremental generate-and-test strategy; meta_qsort, a meta-interpreter running Warren's qsort; nand, a logic synthesis program using branch-and-bound search; simple_analyzer, a flow analyzer analyzing Warren's qsort; poly_10, which symbolically raises a polynomial to the tenth power; chat_parser, which parses a set of English sentences; boyer, an extract from a Boyer-Moore theorem prover; and peep, the VLSI-BAM peephole optimizer processing meta_qsort. Further information about many of the benchmarks may be found in [12].

Table 6.1 compares the performance of the VLSI-BAM processor to that of two other Prolog systems. A more complete comparison of the performance of various Prolog systems can be found in [34, 32]. The results for VLSI-BAM are simulated assuming a 20 MHz clock and include overhead due to cache misses [6]. A clock speed of 20 MHz is used because it is the speed at which several VLSI-BAM chips successfully executed all the benchmark programs. The simulated system has 128 KB instruction and data caches. The caches are direct mapped and use a write back policy. They are run in warm start, that is, each benchmark is run twice and the results of the first run are ignored. Cache effects are significant only for the last six programs in Table 6.1. The cache overhead is greatest for simple_analyzer, and poly_10; for these programs the overhead ranges from 11% to 38%. For meta_qsort and chat_parser the overhead is less than 3%.

The KCM [2], one of the fastest WAM implementations, has a relatively large amount of specialized hardware to execute a WAM-like instruction set efficiently, whereas the VLSI-BAM processor uses modest hardware to support an optimizing compiler. We find that the speed advantage of the VLSI-BAM over the KCM is equal to or greater than the cycle time ratio.

Although the same compiler is being used for both the SPARC and VLSI-BAM machines, the VLSI-BAM outperforms the SPARC for several reasons. First, there is the improvement due to specialized hardware support (this accounts for between 30 and 40%), but the majority of the difference in performance is due to the SPARC-station 1+'s relatively slow load and store instructions. Because Prolog programs heavily use memory loads and stores, a slow memory system will have a dramatic effect on the performance.

A common measure of Prolog speed is logical inferences per second (LIPS). In general this quantity is ambiguous; however, it is well defined for the naive reverse benchmark. The VLSI-BAM processor correctly executes naive reverse at 27.5 MHz[10] giving a measured performance of 3.37 million LIPS.

7. CONCLUSIONS

The primary goal of our research has been to determine a minimal set of extensions to a general purpose architecture necessary for achieving high performance logic programming. At the same time, however, performance of the general purpose architecture has not been compromised. When tags are placed in the most-significant end of the word, we have identified tagged-immediate support, segment mapping, double-word memory bus, special logic for fast branch on tag, and multi-cycle instruction support as important Prolog specific features. This hardware support gives a 64% performance benefit and costs 13% of the VLSI-BAM chip area. When tags are placed in the least-significant end of the word, then double-word memory bus and fast branch on tag logic continue to be very important. In this case, specialized hardware support gives a 38% performance benefit. Even when we use an improved compiler, specialized instruction set support gives about a 32% performance benefit.

Our special instructions for trailing and unification of atoms are of marginal performance benefit. We conclude that branches with three or more directions are not

[10] The instruction sequences which limit the clock speed to 20 MHz do not occur in naive reverse.

effective for Prolog, especially as compilers improve. Our measurements, however, justify the utility of multi-word memory loads and stores, fast branch-on-tag instructions, push instructions, and tagged arithmetic. Such instruction set support would not only improve Prolog performance, but would also be advantageous to the implementation of other dynamically typed languages.

8. ACKNOWLEDGEMENT

We would like to thank the following people for their invaluable contributions, without which the VLSI-BAM would not have become a reality. Joan Pendleton designed the VLSI-BAM's microarchitecture and VLSI circuit design and layout. Charlie Burns provided the VLSI-CAD tools used in the design of the VLSI-BAM processor. William Bush, Ralph Haygood, Joan Pendleton, and Tep Dobry made substantial contributions to the instruction set design of the VLSI-BAM. Ralph Haygood authored the Aquarius Prolog run-time library and VLSI-BAM assembler. Georges Smine and Ketan Bhat helped with the design and simulation of the VLSI-BAM cache board.

We also acknowledge the numerous discussions with the other members of the Aquarius project, especially Tom Getzinger, Ashok Singhal, Hervé Touati, and Vason Srini. Special thanks to Jim Testa for encouraging us to start a new Prolog chip design.

We wish to thank Zycad Corporation for the use of their N.2 hardware simulation tools that simplified the task of simulating the microarchitecture. Equipment and other support for the project was provided by SUN, DEC, and TRW.

REFERENCES

1. J. Beer. The occur-check problem revisited. *Journal of Logic Programming*, 5(3):243–261, September 1988.
2. H. Benker, J. M. Beacco, S. Bescos, M. Dorochevsky, Th. Jeffre, A. Pohimann, J. Noyé, B. Poterie, A. Sexton, J. C. Syre, O. Thibault, and G. Watzlawik. KCM: A knowledge crunching machine. In *16th Annual International Symposium on Computer Architecture*, pages 186–194, May 1989.
3. P. Bitar and A. M. Despain. Multiprocessor cache synchronization, issues, innovations, evolution. In *13th Annual International Symposium on Computer Architecture*, pages 424–433, June 1986.
4. G. Borriello, A. R. Cherenson, P. B. Danzig, and M. N. Nelson. RISCs vs. CISCs for Prolog: A case study. In *Second Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 136–145, October 1987.
5. R. Carlson. The bottom-up design of a Prolog architecture. Technical Report UCB/CSD 89/536, University of California, Berkeley, May 1989.
6. Michael Carlton, Joan Pendleton, Bruce K. Holmer, Barton Sano, and Alvin M. Despain. Cache and multiprocessor support in the BAM microprocessor. In *Fourth Annual Symposium on Parallel Processing*, April 1990.
7. A. Chang and M. F. Mergen. 801 storage: Architecture and programming. *ACM Transactions on Computer Systems*, 6(1):28–50, February 1988.
8. S. K. Debray and D. S. Warren. Automatic mode inference for Prolog programs. In *1986 Symposium on Logic Programming*, pages 78–88, September 1986.

9. T. P. Dobry. *A High Performance Architecture for Prolog.* Kluwer Academic Publishers, 1990.
10. Michel Dorochevsky, Jacques Noyé, and Olivier Thibault. Has dedicated hardware for Prolog a future? In *Proceedings of Processing Declarative Knowledge (PDK '91)*, pages 17–31, 1991.
11. A. Harsat and R. Ginosar. CARMEL-2: A second generation VLSI architecture for flat concurrent Prolog. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 962–969, December 1988.
12. R. Haygood. A Prolog benchmark suite for Aquarius. Technical Report UCB/CSD 89/509, Computer Science Division, University of California, Berkeley, April 1989.
13. John Hennessy, Norman Jouppi, Steven Przybylski, Christopher Rowen, and Thomas Gross. Design of a high performance VLSI processor. In *Third Caltech Conference on VLSI*, pages 33–54, 1983.
14. Bruce K. Holmer and Alvin M. Despain. Viewing instruction set design as an optimization problem. In *24th Annual Workshop on Microprogramming and Microarchitecture (MICRO-24)*, pages 153–162, 1991.
15. Bruce K. Holmer, Barton Sano, Michael Carlton, Peter Van Roy, Ralph Haygood, William R. Bush, Alvin M. Despain, Joan M. Pendleton, and Tep Dobry. Fast Prolog with an extended general purpose architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 282–291, 1990.
16. M. G. H. Katevenis. *Reduced Instruction Set Computer Architectures for VLSI.* MIT Press, 1985.
17. Mark Korsloot and Hans Mulder. Sequential architecture models for Prolog: A performance comparison. *New Generation Computing*, 9:201–209, 1991.
18. J. W. Mills. *LIBRA: A High-Performance Balanced Computer Architecture for Prolog.* PhD thesis, Arizona State University, December 1988.
19. J. W. Mills. A high-performance LOW RISC machine for logic programming. *Journal of Logic Programming*, 6(1&2):179–212, January/March 1989.
20. H. Nakashima and K. Nakajima. Hardware architecture of the sequential inference machine: PSI-II. In *1987 Symposium on Logic Programming*, pages 104–113, August 1987.
21. J. Pendleton, S. Kong, E. Brown, F. Dunlap, C. Marino, D. Ungar, D. Patterson, and D. Hodges. A 32-bit microprocessor for Smalltalk. *IEEE Journal of Solid State Circuits*, SC-21(5):741–749, October 1986.
22. G. Radin. The 801 minicomputer. In *Proceedings Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 39–47, March 1982.
23. Barton Sano. Performance vs. cost of the BAM. Technical Report TR-89-01, Advanced Computer Architecture Laboratory, University of Southern California, December 1989.
24. K. Seo and T. Yokota. Design and fabrication of Pegasus Prolog processor. In *VLSI 89*. North-Holland, 1989.
25. V. P. Srini, J. V. Tam, T. M. Nguyen, Y. N. Patt, A. M. Despain, M. Moll, and D. Ellsworth. A CMOS chip for Prolog. In *Proceedings of the International Conference on Computer Design*, pages 605–610, October 1987.
26. P. Steenkiste and J. Hennessy. Tags and type checking in LISP: Hardware and software approaches. In *Second Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 50–59, October 1987.
27. L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.
28. Andrew Taylor. *High Performance Prolog Implementation.* PhD thesis, University of Sydney, June 1991.
29. H. Touati and A. Despain. An empirical study of the Warren abstract machine. In *1987 Symposium on Logic Programming*, pages 114–124, August 1987.

30. D. M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System.* MIT Press, 1987.
31. Peter Van Roy. An intermediate language to support Prolog's unification. In Lusk and Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 1148–1164. MIT Press, October 1989.
32. Peter Van Roy. 1983–1993: The wonder years of sequential Prolog implementation. *Journal of Logic Programming*, 19/20:385–441, May/July 1994.
33. Peter Van Roy, B. Demoen, and Y. D. Willems. Improving the execution speed of compiled Prolog with modes, clause selection, and determinism. In *TAPSOFT '87*, Lecture Notes in Computer Science, 250, pages 111–125, March 1987.
34. Peter Van Roy and Alvin M. Despain. High-performance logic programming with the Aquarius Prolog compiler. *Computer*, 25(1):54–68, January 1992.
35. Peter L. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California, Berkeley, December 1990. Available as U. C. Berkeley Computer Science Division technical report no. UCB/CSD 90/600.
36. D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, October 1983.
37. M. Yokota, A. Yamamoto, K. Taki, H. Nishikawa, and S. Uchida. The design and implementation of a personal sequential inference machine: PSI. *New Generation Computing*, pages 125–144, 1983.