Preliminary Plan for BALM-to-LITTLE Translator. J. Schwartz

1. Introduction

The debugging of the LITTLE-written, SRTL-compatible BALM interpreter which will initially support the new SETLB system is now far advanced. However, in order to avoid substantial losses in BALM compile efficiency, and possibly significant losses in SETLB execution efficiency as well, we will undoubtedly require a SRTL-compatible BALM translator as a replacement for the interpreter. This is, of course, an SRTL-compatible version of the present (R. Paige) BALM translator. This newsletter will outline a plan for such a translator. The overall plan is to go BALM to LITTLE-internal (VOA entries and associated tables), though for initial debugging (and discussion) BALM to LITTLE-source is probably better. The scheme to be proposed suggests certain extensions to the present system of peephole optimizations applied during LITTLE code generation, and also some modifications in the handling of index registers, with reservation of a few B-registers for global BALM pointers. Stack-pointer calculations can be systematically optimized. Certain other peephole optimizations applicable during BALM to LITTLE translation also deserve consideration.

2. Statistical information on BALM code.

To estimate the space costs and speed efficiency of any BALM translation scheme, basic statistical information on the static and dynamic frequency of the various BALM machine instructions are required. Here are such figures, for the BALM compiler and a short run of it, supplied by Stephanie Brown.

	BALM Instruction	Length (Bytes)	Static Freq.	Dynamic Freq.
1.	ID2→STACKTOP	3	960	10
2.	STACKPTR PUSHUP	1	530	9
3.	I1→STACKTOP	2	430	3
4.	I2→STACKTOP	2	410	1
5.	STACK(I1) → STACKTOP	2	400	8
6.	ARG(I1) → STACKTOP	2	370	12
7.	CALL	2	360	4
8.	IDENTIFIER FROM INT.	1	330	1
9.	JUMP L2	3	300	3
10.	POP I1	2	290	4
11.	GSTORE I2	3	280	2
12.	NIL → STACKTOP	1	240	3
13.	STACKTOP → STACK(I1)	2	240	5
14.	MAKE LIST	3	220	0.1
15.	JUMP FALSE L2	3	210	8
16.	STKPTR PUSHUP WITH SAVE	1	160	-
17.	MAKE PAIR	1	140	1
18.	HEAD	1	110	4
19.	TAIL	1	100	3
20.	RETURN	1	100	4
21.	OTHERS	2	380	15
	Total:		3200 insts	100K insts

The 20 most frequent instruction account for 85 % of the instructions statically and about the same percentage dynamically.

We shall suppose that the LITTLE code generator is modified so as to recognize the following global BALM quantities, and keep them in B-registers during the execution of BALM code: STACKPTR, ARGBASE, VARBASE.
Moreover, we assume that array references of the form A(index + const) are optimized by the absoprtion of a constant, and compiled as

LOAD[A + const](index) .

Similarly for array stores.

Up to 3 stack positions can be held in the global LITTLE quantities INP1, INP2, and RESULT, for which full-length registers may be reserved. Multiple versions of certain very short executive routines may be provided so as to make it unnecessary to move arguments among these registers. During the generation of each LITTLE instruction the BALM to LITTLE translator will be aware of the loading of the 3 special registers INP1, INP2, and RESULT, and also of a 'correction' applicable to the STACKPTR value. The quantities TRUE and NIL can be carried in 'represented by nonzero', 'represented by zero' etc. forms. This allows us to compile certain of the BALM instructions efficiently in-line; the remaining instructions will be compiled by loading their arguments (if necessary) and return-jumping to an executive routine. The instructions which it should be advantageous to compile in-line are as follows:

JMPT	JMPF	JMP	NUM2	GLOB	GSTORE
LBL	NUM1	VAR	VSTORE	ARG	ASTORE
POP	NUM3	NEGATE INT	IPOSQ	IZEROQ	IDENTQ
V[I]	V[I]=X	NOT	SETSX	TRUE	NIL
SETSTK	SIMTYPE				

3. Proposed translations. Anticipated code-expansion factor.

The above list includes all of the 20 most common instructions, with the exception of CALL, IDENTIFIER FROM INT., MAKE LIST, PAIR, HD, TAIL, RETURN; it includes at least 60% of the instructions counted statically, and 70% of the instructions counted dynamically. The following table shows typical LITTLE translations for each of the instructions to be translated in line, and gives the number of bytes of BALM machine and optimized 6600 code which will result.

	BALM Instruction	Bytes	Translation	Bytes
1.	GLOB ID2	3	INP = SYMBTAB(ID2)	4
2.	SETSTK	1	STP = VB	2
3.	NUM1	2	INP = K	4
4.	NUM2	3	INP = K	4
5.	VAR I1	2	INP = STK(VB + I1)	4
6.	ARG I1	2	INP = STK(AB+I1)	4
9.	JMP L2	3	JP L2	4
10.	POP I1	2	STP = STP - K	4
11.	GSTORE ID2	3	SYMBTAB(ID2) = RESULT	4
12.	NIL	1	INP = UNDEFWD	4
13.	VSTORE I1	2	STK(VB+I1) = RESULT	4
15.	JMPF L2	3	ZEROJP L2	4
16.	SETSX	1	STK(VB+NV+1)=RESULT;	
			STP = VB+NV+1	6
Others:	JMPT L2	3	NZJP L2	4
	LBL L2	3	INP = PRECALC. CONSTANT	6
	ASTORE	2	STK(AB+I1) = RESULT	4
	NUM3	4	INP = PRECALC. CONST	12
	NEGATE	1	INP = INP.XOR.SBIT	6
	IPOSQ	1	INP = ESIGN INP	6
	IZEROQ	1	INP = EMAG INP	6
	IDENTQ	1	RESULT = INP1.XOR.INP2	
			.AND.MASK	6
	V[I]	1	RESULT=HEAP(INP1+INP2)	6
	V[I] = X	1	HEAP(INP1+INP2) = RESULT	6
	NOT	1	RESULT = INP.XOR.NILWD	6
	TRUE	1	RESULT = TRUE	4
	SIMTYPE	1	RESULT = IN1.XOR.INP2	
			.AND.MASK	8

Concerning the V[I] and V[I] = X instructions, note that the translation shown can only be used if we make the assumption that LITTLE will automatically truncate integers used as array indices to the size of total memory. If this is false, the translation must include a masking operation, becomes 12 bytes long, and the operation would probably be done offline rather than inline. Note also that the suggested translation involves no type-checking and no out-of range checking. These might be provided as part of a special 'debug' option.

Other instructions will be translated as

CALL EXECROUT,

where EXECROUT is an appropriate executive routine. Since this may waste half a word when compiled to the 6600, we estimate its length at 6 bytes. Thus space blowups for the common instructions compiled off-line are

	Instruction	Freq.	Bytes	Translated Bytes
7	CALL	360	2	6
8	IDENTIFIER FROM INT	330	1	6
14	MAKE LIST	220	3	6
17	MAKE PAIR	140	1	6
18	HEAD	110	1	6
19	TAIL	100	1	6
20	RETURN	100	1	6

The roughly 13,000 bytes of BALM machine code represented in the preceding tables should expand into approximately 30,000 bytes of compiled 6600 code, an expansion factor of roughly 2.5/1. Execution speeds should be rather good, essentially those attained by the present BALM translator.

The CALL, TEST LOOP, HEAD, and TAIL operations deserve additional remark, the second of these not for static but for dynamic frequency of occurrence. The BALM calling sequence for functions of two arguments will typically have a form such as

```
GLOB I2, VAR I1, GLOB I2, CALL 3
```

and therefore require 10 bytes. The LITTLE translation of this would be something like

```
INP1 = SYMT(I2), INP2=STK(STP+I1), RESULT=SYMT(J2),
    CALL KALLRIN1IN2RES
```

requiring 16-22 bytes and not involving any great amount of waste motion; the KALLR routine can perform linkage and all required register saves. Calls to most primitives will require 14 bytes, with inputs returned in standard registers. Compilation of the TEST LOOP instructions might usefully be special-cased to detect loops with positive constant starting point and increment, and with loop invariant upper limit. (Note however, that this requires at least a small amount of global program analysis.) Such loops could be compiled in a manner substantially more efficient than results from straightforward compilation, which gives code involving calls to the SRTL integer addition routine.

The 'obtain head' and 'assign head', and the corresponding 'tail' routines can be compiled inline if type-validation is omitted and if we make the assumption needed to justify inline treatment of the 'obtain tuple component' and 'assign tuple component' operations.

4. Concerning BALM code compression and a microcoded BALM machine

The BALM machine code for a typical source fragment such as A = B + C will typically be something like

```
GLOB B, ARG C, ADD, STOREVAR A
```

which is 3 + 2 + 1 + 2 = 8 bytes long. Greater density can be

achieved by passing to a 3-address style of code in which each operation is followed by the list of its arguments, which are flagged (in 2 bits) as being either local variables, arguments, immediates, or globals (the latter requiring 2 bytes). In this style, A = B + C would compile as

ADD(GLOB B ARG C) → VAR A ,

requiring 1 + 2 + 1 + 1 = 5 bytes . Thus BALM code could be compressed to about 60% of its present size. Suitable hardware could execute the compressed code as rapidly as the existing BALM machine code, and even as fast as its translation.

Note that this degree of compression, which is probably close to the maximum attainable, comes from the separation of the environment of BALM processes into the following subparts.

i. Local variables, addressed by one byte
ii. Arguments, addressed by one byte
iii. Global variables, more numerous, and addressed by two bytes
iv. Heap space, requiring longer addresses.

Local variables and arguments have a pattern of usage allowing them to be concentrated at the top of the BALM stack.

An optimizing BALM translator for a microcoded multi-register machine could probably keep quantities of types i, ii, and iii in registers, and fit most instructions involving only such quantities into the time between stores and loads of heap quantities. Since heap references are dynamically about 14% of BALM code, the maximum execution speed (BALM instructions) attainable with this kind of adapted architecture is therefore about 5-6 times the basic heap-memory cycle time, or about 5 mips running against a 1 micro-second bulk memory. Paging to faster buffer memory might double or triple this. The micro machine should have at least five times the bulk memory speed, and with a fast buffer memory probably at least ten times this speed.

The microinstructions desirable for the SRTL executive code are different from those desirable for BALM simulation, but the same gross conclusions concerning attainable speeds are probably valid. SETL instruction rates of 2-5 mips based on a 50 mip micro-machine appear plausible.