Regular Paper

Hardware Support and Code Generation for Dynamic Range Checking in C

PAUL SPEE[†*] and EIICHI GOTO[††]

Runtime checking in general and runtime array subscript checking in particular is considered to be very costly in terms of execution speed. We will show that runtime checking and optimal execution are not necessarily exclusive. We present a novel memory protection scheme called *BL-addressing* which allows range checking to be done in parallel with the memory access, reducing or removing overhead caused by range checking. We describe the implementation of the BL-addressing in a C compiler. At the same time we suggest that a small change in the ANSI C language definition is necessary to successfully implement array subscript checking.

1. Introduction

To improve program reliability and to protect the user from unintentional programming mistakes, a program can be checked in two ways. *Static checking* takes place during program compilation while *dynamic checking* takes place during program execution. Of the various forms of static checking, *static type checking* is the most important. Static type checking can be done in languages which are statically typed. In languages such as Lisp, data types are associated with objects during runtime and type checking must be done dynamically. Type checking verifies whether the type of the operands matches the type required by the operator; type coercion may be optionally done. Static checking has important advantages. The programmer is notified of the error, not the user. The check is only performed once, not each time the program is executed or worse, many times during program execution.

However, there are conditions which simply cannot be checked during program compilation. When an array is indexed, the value of the index is not known until the program is executed. If the index lies outside the array boundaries, incorrect data can be read or data can be stored in unrelated locations. As a result the execution can either unexpectedly abort, or worse, the program can complete 'correctly,' but with incorrect results. This can be anywhere from inconvenient to fatal; consider for example banking transaction systems, satellite control, and life support systems.

In contrast to Pascal and Ada, which require strict runtime checking, C is meant to be a small and efficient system programming language. Kernighan and Ritchie write, "*C retains the basic philosophy that programmers know what they are doing; it only requires that they state their intentions explicitly*".[15]

Our experience with students programming in C showed that most of the programming errors were caused by illegal access through pointers and illegal array access. Hoare once compared the evil of pointers with the evil of gotos.[13]

In this paper we will present a new memory protection scheme. The memory protection scheme was not originally designed to provide hardware assistance for runtime array checking. Only much later were its features implemented in a C compiler to provide optional runtime checking. The runtime checking adds only a small overhead; actually in some cases code can run even faster using this addressing scheme. Secondly, we would like to make a case for adding optional runtime checking to C. Using optimization techniques such as *common subexpression elimination, code motion, induction variable elimination*, etc., such runtime checking can be implemented efficiently. Finally, we describe here the implementation of code generation and code optimization for BL-addressing in the C compiler; it can be equally well applied to Pascal, Fortran, Ada, etc.

[†] Research Development Corporation of Japan (ERATO)
[††] Faculty of Science, University of Kanagawa
[*] Currently, Unix System Laboratories

2. Hardware Assist for Range Checking

Several architectures provide hardware assist for dynamic range checking in one way or another. First, we will describe architectural support for dynamic range checking which can be found in other architectures. We will then describe the *BL-addressing* scheme which is employed by FLATS2[14] a cyclic pipeline shared memory MIMD processor.[21]

2. 1 Trap Instruction

While it would be possible to implement range checking using compare, branch and trap instructions, most CISC processors provide special instructions to aid range checking. The VAX architecture[8] provides the index instruction which does the index calculation and incorporates range checking. The 68000 series processor[16] includes two instructions for range checking. The chk instruction checks for the contents of a data register to be in the range of 0 and a value specified in the operation. The chk2 instruction checks a register to be within the range specified by a lower and upper bound in memory. The Intel 80286/80386 provide a special bound instruction. The IBM 801 project[20] showed that runtime range checking could be implemented with little overhead using a special *compare and trap* instruction and an optimizing compiler.[4],[16]

2. 2 Descriptors

In descriptor-based architectures, a data segment is described by a descriptor which specifies both the starting address and the length of the segment. Each access to the segment goes through its segment descriptor. Well known examples of descriptor-based architectures are the Burroughs B6700[19] and the ICL 2900.[7]

Ada renewed the interest in descriptor-based architectures and new approaches were proposed.[6] Bishop stated the problems associated with descriptors: a) descriptors control the size of items, b) setting up descriptors is very costly, c) descriptors mean unnecessary checks, and d) descriptors are confused with indirect addresses.

Hill suggested the inclusion of two extra registers in the ALU. A special instruction *load ranges* would load the low and high ranges into those registers. During the subsequent operation, the result of the ALU is checked against the range registers.[12]

2. 3 Capabilities

Capabilities[10],[22] are an extension of descriptors. While descriptors refer to a memory segment, capabilities refer to an object to which the owner of the capability has certain access rights. A capability represents the right to access (read/write) the data, destroy the capability or copy the capability. Each capability is unique in that it contains an object identifier. Two ways exist to protect capabilities. The first way uses tags to identify capabilities, the second way uses separate capabilities and segments.

An example of a capability machine is the IBM System/38.[5] The System/38 has three provisions for securities: a) capability based addressing, b) objects called User Profile determine the protection domain of a process and contains the access right to system objects, and c) processes which are isolated from each other. Access to objects is done using tagged pointers. The System/38 provides four types of capabilities: System Pointers to address objects, Space Pointers to address a byte location in a space object, Data Pointers, which are like Space Pointers but contain attributes of the data they locate, and Instruction Pointers, which act as label values for variable branches within a program. Each capability consists of 16 bytes.

The SWARD machine[18] implements both tagged storage and capability based addressing. The machine recognizes 15 data types; 10 primitive data types and five complex data types (array, parameter list, relocatable, structure, user).

Examples of capability-based machines which use segmentation to distinguish between capabilities and data are the Plessey System 250, the Cambridge CAP computer, and the Intel iAPX 432.

2. 4 BL-addressing

The BL-addressing scheme was initially intended as a memory protection scheme, not a range checking scheme. The scheme was implemented in the FLATS2,[14] a two instruction stream cyclic pipeline shared memory processor.[21]

Most modern architectures often support paging, segmentation or a combination of both virtual memory schemes. Page or segment tables contain a description of the information needed for the address translation from a logical address to a physical address. The table entries contain information such as access rights and other protection information. The VAX page table entry[8] has a protection field which describes the read/write access rights for each of the following modes: kernel, executive, supervisor, and user.

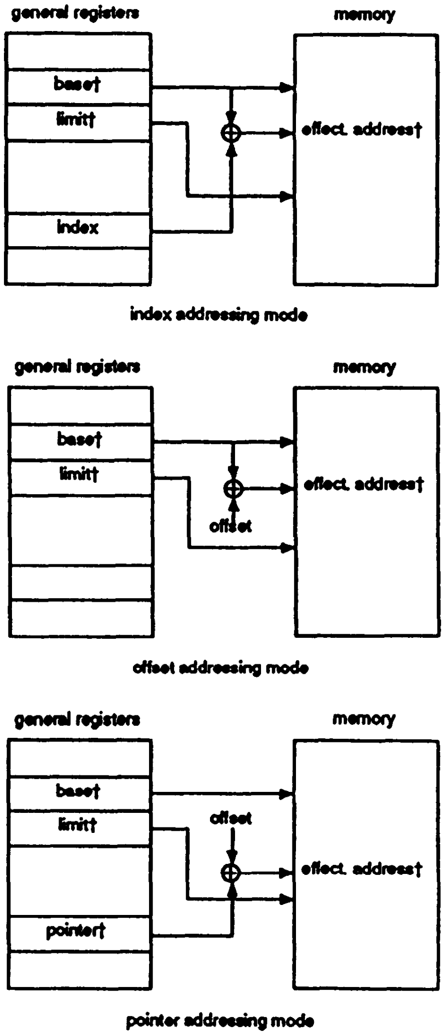
Because of the memory access time requirements, the FLATS2 memory system could not be implemented using such an elaborate scheme as used by the VAX (and many other architectures). The memory system of the FLATS2 is implemented as a very large cache. Constraints do not allow the addition of a memory protection scheme in the memory system. Instead it uses a memory protection scheme called the *B* (*ase*) *L* (*imit*) *-addressing* scheme. The BL-addressing scheme can be considered a form of descriptor addressing. Two general purpose registers, one containing the address of the start of a memory segment* and the other containing the address of the end of a memory segment, form together a descriptor called BL-register pair. Each memory access requires the specification of a BL-register pair; the effective memory address is checked against the base and limit address. We will discuss BL-addressing in more detail in the following paragraphs.

BL-register pair

A word consists of 33 bits, 32 data bits and a one bit tag. The tag specifies whether the word is to be treated as a 32 bit integer or a 32 bit address. The tag is therefore refered to as the *address tag*. Both memory and the general purpose registers are tagged. Each register can either contain a value or an address. If an even and odd numbered general purpose register both contain a valid address, they can be used as a *base* and *limit* register pair.**

Effective address calculation

Depending on the memory addressing mode, a general purpose register is either treated as the base or limit register of a BL-register pair, index register or pointer register. FLATS2 has three general purpose addressing modes (see **Fig. 1**).



† address tag set

Fig. 1 Addressing modes.

In the index mode, the contents of the base register (must contain a valid address) and the contents of the index register (must contain an integer) are added to form the effective address. In offset mode, the contents of the base register and an immediate offset are added to form the effective address. In addition, in pointer mode, the effective address is formed by adding the contents of a pointer register (must contain a valid address) and an immediate offset. For a more detailed description and the assembly language notation of the FLATS2 addressing

* Although we use the word segment here, it does not imply the memory space is segmented; the FLATS2 memory space is linear.

** A BL-register pair is specified by its base register. If the register is an odd numbered register, the previous even numbered register is the limit register.

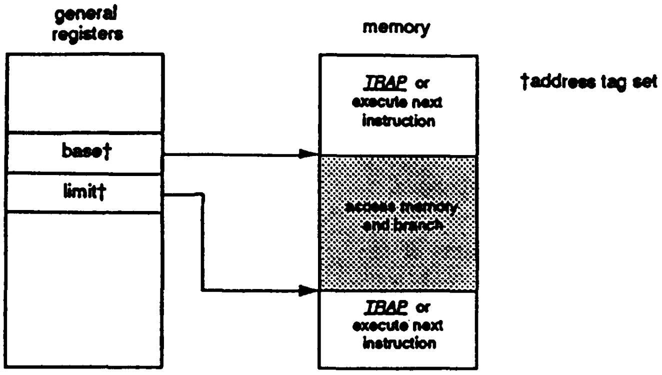


Fig. 2 BL-range checking.

modes, refer to Appendix A.

BL-range checking

For each memory reference a BL-register pair must be specified. The effective address is checked against the base and limit registers. The memory access fails if the base, limit or effective address is not a valid address or if the effective address lies outside the range specified by the base and limit registers. If the memory access succeeds, a branch to a specified location occurs; otherwise, the next instruction is executed (**Fig. 2**). Each instruction which can have a memory operand has a field specifying the branch address.* If the specified address is the address of the next instruction, a *memory address error* trap occurs when the base, limit or effective addresses are not valid addresses or a *BL overflow error* trap occurs when the effective address lies outside the boundaries specified by the base and limit registers. The 'branch to' address can be used to implement tight loops by using the BL-range checking control as an end of loop test. If we would have used the branch field to specify the address to 'branch to' on failure, we would not have been able to successfully make use of the BL-range checking as an end of loop test, because we would require an additional branch instruction to branch to the beginning of the loop. Under the assumption that a loop is executed more than once, this would result in an additional instruction being executed for each iteration.

Address creation

Like capability based systems, only instructions related to address calculation can set the address tag. All other instructions clear the address tag. This protects the user from forming an illegal address. Furthermore, new addresses (and subsequent BL-pairs) can only be derived from existing valid addresses and BL-register pairs. This is important to guarantee that the user can never form an illegal address and thus can address the memory outside the assigned address space.

Using such instructions as the load effective address instruction lea, new addresses can be created and these new addresses can then form a new BL-register pair. If the calculated effective address falls outside the specified BL-register pair, the address tag is cleared. Consequently, the resulting value cannot be used as an address nor as a base or limit address in a BL-register pair.

Discussion

While the IBM System/38 supports tagged capabilities and has the concept of a Space Pointer which may be adjusted to point to any byte within the space object, it would not be correct to consider BL-addressing a form of capability addressing. Concepts such as objects, capability lists, domain switching are all missing from the BL-addressing scheme. In our opinion, the BL-addressing should be considered a form of descriptor addressing. However, the problems raised by Bishop do not apply to the BL-addressing scheme. Using BL-addressing does not necessarily add to the execution time. A special instruction exists to create a BL-register

* The Crisp processor uses inline branch fields for branch folding,[9] that is, the instruction cache includes a branch field for each instruction.

pair in one machine cycle; this is no more than calculating an effective address (see Table 1 for an example). The check of the base and limit is done in parallel with the memory access and therefore does not cause additional overhead. Furthermore, the BL checking is an inherent part of the memory protection scheme; it is always performed.

While Gehringer argues that all aims of tagging can be achieved without hardware tagging,[11] it should be noted that in our scheme, memory protection could not be guaranteed without the assistance of hardware supported tags.

Using descriptors may cause a large overhead if only a few segment registers are provided. It would cause unnecessary reloading of segmemt registers. For this reason, the FLATS2 architecture allows any general purpose register pair to be used as a base-limit register pair. Furthermore, a total of 32 global general purpose registers and 32 local general purpose registers are provided. The local general purpose registers are switched by the call and return instructions. This number of registers is sufficient to keep most BL-pairs in registers.

3. Compiling for Range Checking

In this section we will discuss the implementation of range checking in C using the BL-addressing scheme. We will first discuss some of the general problems associated with the implementation of arrays in C before going into details on the actual implementation. We will see that for each array access a lot of additional code is generated. However, using various optimization techniques[2] such as *common subexpession elimination, loop invariant code motion, strength reduction*, and *induction variable elimination*, it is possible to reduce the overhead incurred by the runtime range checking.

3.1 Arrays in C

The support for arrays in C has always been minimal. By equating an array to be equivalent to a pointer to the first element of an array (Ref. 1, § 3.3.2.1), the semantic level of arrays is lowered. This can easily be seen from the fact that when a two dimensional array is passed as an argument to a subroutine, the programmer has to explicitly do the array index calculation. Arrays in C are second class citizens; this is the main reason why C never succeeded in attracting the scientific community. Not only is most of the scientific code written in FORTRAN (the dusty deck problem), but lack of sufficient support for arrays makes efficient compilation for vector processors (vectorization) a difficult task.[3]

Inlining of functions would remove some of the problems associated with array parameters, because the information on the array size would become available to the function body. However, dynamically allocated arrays cannot be declared correctly, leaving the problem largely unsolved.

To be able to use range checking with dynamically allocated arrays or array parameters, we allow the declaration of *a pointer to a variable size array*. Unfortunately, declaration of pointers to variable size arrays are **not** allowed by the ANSI C standard. **Example 1** shows the use of pointer to variable size array declarations.

The function is called as

```
dasum(size, array);
```

This construct has several advantages. The base of the array is still represented by an address; existing sources or object files will not be broken. The compiler can choose to ignore it, as it only specifies the size. It is possible to write more readable code (*array) [x][y][z] instead of *(array+sizeof (array[0])/sizeof (array [0][0][0])*x+sizeof (array[0][0])/sizeof (array[0][0][0])*y+z).

3.2 Implementation of Pointers

Before discussing the implementation of array access, we will first discuss our implementation of pointers. Pointers and arrays are closely related; in fact, the C reference manual states that

```
double dasum(n, dx)
register    n;
register double   (*dx)[n];
{
        double      sum;
        register    i;

        for (i = 0; i < n; i++)
           sum += (*dx)[i];

        return sum;
}
```

Example 1 Variable size array parameter.

```
#define     INTS  6

int    iv = 3;
int    *a[INTS] = { &iv, &iv, &iv, &iv, &iv, &iv };
int    x = (int) &iv;     /* this is not an address */

main()
{      int    i;
       int    sum = 0;
       int    **ip = a;

       for (i = 0; i <= INTS; i++)    /* '<=' is wrong */
       {
              sum += **ip++;
       }
       printf("sum = %d\n", sum);
}
```

Example 2 Illegal pointer access.

the array access array [index] is identical to •(array+index). In principle, it is possible in C to assign any value to a pointer and use this pointer to access memory. As long as the pointer points to the user data space, the pointer can be used to read or write the location it points to (within the memory alignment constraints). Whether the result is actually what the user intended, is sometimes doubtful. As the C pointer can point to any location within the user's data space, it is very easy to corrupt data. For example, the use of an integer as a pointer (casting or argument passing) or during pointer addition, can result in undersirable effects. **Example 2** shows how a simple programming error can result in the incorrect execution of the program and thus results in the incorrect output. The most likely result will be 'sum=21', e. g. seven times the contents of iv, but which is not correct, because the seventh address is not a legal address. Executing this program on a Sun 3 results in the incorrect result, while executing this program on the FLATS2 results in an exception.

The above problem can easily be solved by implementing pointers as addresses (address tag is set) and implementing non-pointers as integer/floating point values (address tag is cleared). This provides a rudementary form of dynamic type checking.

In the intermediate code generated by the C compiler, a pointer is differentiated from an integer. The compiler makes sure that all operations on pointers expressed in intermediate code will follow the rules of pointer arithmetic. A pointer plus or minus an offset results in a pointer. A pointer minus a pointer results in an integer. A pointer plus a pointer is undefined. An integer minus a pointer is undefined.

Pointer calculation uses a BL-register pair which represents the user address space and has been provided by the operating system.

During code generation, the compiler knows exactly when to emit code for integer arithmetic or pointer arithmetic. Pointer arithmetic $(p+i)$ is implemented using the load effective address instruction lea bl:$i(p)$,vr. If $(p+i)$ is a legal address within the area as specified by the BL-pair bl, it is transferred to the destination register vr. Casting from integer to pointer is done using the set address instruction. For example, the cast $p=$(char*) i is implemented by seta bl:(i), p. The value of i is transferred to p. If the value of i is within the range of area specified by the BL-pair bl,* the address tag of p is set. In this case, the value of p is a legal address and can be considered a pointer.

The cast $i=$(int) p is implemented as add p, # 0, i (all non-address calculation instructions automatically clear the address tag). All valid pointers (address tag set) can be used to access memory. If the memory is referenced using a non-address (address tag is cleared), a *memory*

* In this case, the BL-register pair contains the lower and upper bound of the user addressable data space and is provided by the operating system. This value is loaded into a fixed global register pair and is known to the compiler.

address error trap occurs. Executing the previous example on the FLATS2 results in a trap when using the contents of the variable x as a pointer. As long as no casting is used, this runtime checking does not increase execution time.

3.3 Implementation of Arrays

Arrays are implemented as a BL-register pair. The beginning of the array (&array[0]) is specified by the base register, while the end of the array (&array [n]−1) is specified by the limit register. All memory accesses to the array are done using the BL-register pair specifying the array. If the effective address lies outside the array boundaries, a trap occurs. **Figure 3** shows the calculation of the array boundaries and the indexing of the array.

In the previous example, the array is a one-dimensional array. The concept is easily extended to multi-dimensional array addressing. For each dimension, we calculate a BL-register pair. For example, in addressing a three-dimensional array a [i][j][k], we first calculate the boundaries (BL-register pair) of the three-dimensional array a. Using these boundaries, we can calculate the boundaries of the two dimensional plane a[i] (see **Fig. 4**). A trap will occur if the index i is not valid for array a. Next, we calculate the boundaries of the one-dimensional row a[i][j] using the BL-register pair specifying a[i] (see Fig. 4). A trap will occur if index j is not valid. Finally, we can address a [i][j][k] using the BL-register pair of a [i][j] with one of the addressing modes. To summarize, in accessing a multi-dimensional array, first the array boundaries are calculated. Subsequent boundaries are calculated for high order indexing. Finally, the row boundaries are calculated. This BL-register pair can then be used in accessing the row elements. When less concerned with the correct value of the individual indexes than that the accessed location belongs to the array, we can suffice with the calculation of a single BL-register pair.

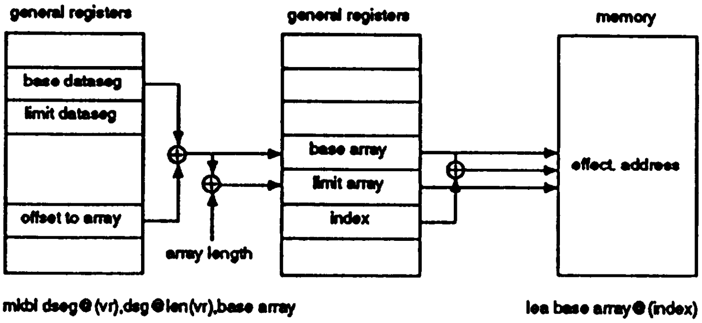


Fig. 3 Array indexing.

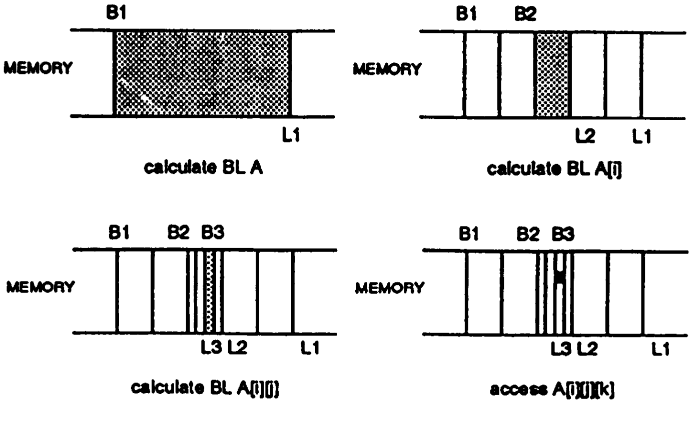


Fig. 4 Indexing a three-dimensional array.

3.4 Code Generation for Array Access

Usually, the intermediate code generated for an array access consists of the calculation of the base address of the array (pointer) and the calculation of the offset into the array and finally the address of the element of the array, consisting of the base address plus offset. For those arrays whose size is known, our compiler optionally generates range checking code. It does this by calculating the base address and limit address of the array and storing the result in a BL-register pair. By taking the base register and treating it as a pointer to the array, the intermediate code generated for the offset calculation and element address calculation is equivalent to the general case. During code emission the base address is recognized to be the base register of a BL-register pair, not a pointer register and as a result the correct memory operand is generated to allow the range checking. The size of statically allocated arrays is known, but the size of dynamically allocated arrays or array parameters is not known. If we would limit the use of range checking to statically defined arrays, we would lose most of the important opportunities.

Table 1 shows that the overhead for calculating the base and limit registers is not necessarily large. In indexing a[*i*][*j*][*k*] (a is statically defined), we make use of a special instruction mkbl, which calculates the base and limit registers in one cycle. The total number of cycles executed is identical for both pointer calculation (conventional) and BL-register calculation. Using the BL-register calculation, initially, we require one additional instruction to calculate the boundaries of the array. However, in addressing the array element, we save one instruction.

The BL-pair cannot always be calculated using an mkbl instruction. When it cannot be, two load effect address instructions are required (each lea instruction executes in one cycle). This is necessary when the the two addresses in the mkbl instruction become too complex to be encoded in one instruction. Also, when calculat-

Table 1 Indexing a[*i*][*j*][*k*].

Pointer calculation		BL-register calculation		
		mkbl	**dseg@_a, dseg@_a+3999, vr6**	; bl for a
mul3.l	**vr1, #400, vr4**	**mul3.l**	**vr1, #400, vr4**	; *i* * sizeof(a[0])
lea	**dseg@_a(vr4), vr6**	**mkbl**	**vr6@(vr4), vr6@399(vr4), vr6**	; bl for a[*i*]
mul3.l	**vr2, #40, vr4**	**mul3.l**	**vr2, #40, vr2**	; *j* * sizeof(a[0][0])
lea	**dseg: (vr6)vr4, vr6**	**mkbl**	**vr6@(vr4), vr6@39(vr4), vr6**	; bl for a[*i*][*j*]
mul3.l	**vr3, #4, vr4**	**mul3.l**	**vr3, #4, vr4**	; *k* * sizeof(a[0][0][0])
lea	**dseg: (vr6)vr4, vr6**			; pointer to a[*i*][*j*][*k*]
movw	**dseg: (vr6), vr0**	**movw**	**vr6@(vr4), vr0**	; load element

```
#define     INTS  6
#define     IV    3

int    a[INTS] = {IV, IV, IV, IV, IV, IV };
int    x = IV;

main()
{
       int   i;
       int   sum = 0;
       for (i = 0; i <= INTS; i++)    /* the '<=' is wrong */
       {
              sum += a[i];
       }
       printf("sum = %d\n", sum);
}
```

Example 3 Illegal array access.

ing the BL-pair for dynamically allocated arrays and array parameters, an additional instruction is required to subtract one from the array limit because the test for the limit boundary is inclusive. The following program demonstrates the use of range checking. Normally, no runtime checking code is generated for array indexing, thus the outside boundary condition is undetected. On a conventional architecture, the value printed will most probably be 21. On FLATS2, the result will be a *BL overflow trap* on accessing a[6] (see **Example 3**).

3.5 Optimization of BL-range Checking

In this section we will describe the various optimizations performed to reduce the additional instructions generated to support BL-range checking. As an example, we use the routine dscal() from BLAS (Basic Linear Algebra Subroutines) (see **Example 4**). The routine dscal() scales a vector with a value.

```
dscal(n, da, dx)
register          n;
double            da;
register double   (*dx)[n];
{
      register     i;
      for (i = 0; i < n; i++)
            (*dx)[i] = da * (*dx)[i];
}
```

Example 4 dscal-scale vector.

Figure 5 gives the code including BL-range checking code generated in non-optimized mode. For each access to dx[] (both load and store) code is generated. This is rather costly as it happens inside the loop.

Common subexpression elimination will remove the second calculation of the BL-register pair for array dx[]. Loop-invariant code motion will move the calculation of the BL-pair outside the loop (see **Fig. 6**).

Strength reduction and induction variable elimination will remove the calculation of the offset into the array (multiply instruction). Finally, peephole optimization reduces the compare instruction and the branch instruction to one compare-and-branch instruction. The add instruction, compare instruction and the branch instruction are reduced to a single add-compare-and-branch instruction (see **Fig. 7**).

3.6 Loop Control Optimization

It is possible to use the range checking as an end of loop test by setting up the base and limit registers so that they do not contain the array boundaries, but instead the addresses of the first

```
_dscal:
        movw    sp,fp
        movw    dseg:0(fp),vr0      ; vr0 = n
        movw    dseg:16(fp),vr1     ; vr1 = dx[]
        mul3.l  vr0,#8,vr2          ; vr2 = n * sizeof(double)
        movw    #0,vr3              ; i = 0
L2:
        cmp.l   vr3,vr0             ; while (i < n) do
        bge     L3
        addr    vr2,#-1,vr6         ; vr6 = sizeof(dx) - 1
        lea     dseg:(vr1),vr4      ; base = dx[]
        lea     dseg:(vr1)vr6,vr5   ; limit = dx[] + sizeof(dx) - 1
        mul3.l  vr3,#8,vr6          ; offset = i * sizeof(double)
        addr    vr2,#-1,vr7         ; vr7 = sizeof(dx) - 1
        lea     dseg:(vr1),vr8      ; base = dx[]
        lea     dseg:(vr1)vr7,vr9   ; limit = dx[] + sizeof(dx) - 1
        mul3.l  vr3,#8,vr7          ; offset = i * sizeof(double)
        mov.d   vr8@vr7,S           ; S = dx[i]
        mov.d   dseg:8(fp),P        ; P = da
        mul3.d  P,S,vr4@vr6         ; dx[i] = da * dx[8]
        addr    vr3,#1,vr3          ; i = i + 1
        jmp     L2                  ; continue
L3:
        movw    fp,sp               ; return
        ret
```

Fig. 5 dscal-unoptimized with range checking.

```
_dscal:
      movw    sp,fp
      movw    dseg:0(fp),vr0        ; vr0 = n
      mov.d   dseg:8(fp),S          ; S = da
      movw    dseg:16(fp),vr1       ; vr1 = dx[]
      mul3.l  vr0,#8,vr2            ; vr2 = n * sizeof(double)
      cmp.l   vr0,#0                ; if (n <= 0) return
      ble     L3
      addr    vr2,#-1,vr6           ; vr6 = sizeof(dx) - 1
      lea     dseg:(vr1),vr4        ; base = dx[]
      lea     dseg:(vr1)vr6,vr5 ; limit = dx[] + sizeof(dx) - 1
      movw    #0,vr3                ; i = 0
L2:
      mul3.l  vr3,#8,vr6            ; vr6 = i * sizeof(double)
      mul3.d  S,vr4@vr6,vr4@vr6 ; *(dx+(i*8)) *= da
      addr    vr3,#1,vr3            ; i++
      cmp.l   vr0,vr3               ; while (i < n)
      blt     L2
L3:
      movw    fp,sp                 ; return
      ret
```

Fig. 6 dscal-cse and code motion optimization.

```
_dscal:
      movw     sp,fp
      movw     dseg:0(fp),vr0        ; vr0 = n
      mov.d    dseg:8(fp),S          ; S = da
      movw     dseg:16(fp),vr1       ; vr1 = dx[]
      mul3.l   vr0,#8,vr2            ; vr2 = n * sizeof(double)
      cmp.ble  vr0,#0,L3             ; if (n <= 0) return
      addr     vr2,#-1,vr6           ; vr6 = sizeof(dx) - 1
      lea      dseg:(vr1),vr4        ; base = dx[]
      lea      dseg:(vr1)vr6,vr5     ; limit = dx[] + sizeof(dx) - 1
      movw     #0,vr3                ; i = 0
      mul3.l   vr0,#8,vr0            ; vr0 = n * sizeof(double)
L2:
      mul3.d   S,vr4@vr3,vr4@vr3     ; *(dx+(i*8)) *= da
      ac.bgt   vr0,#8,vr3,L2         ; while ((i += 8) < (n*8))
L3:
      movw     fp,sp
      ret
```

Fig. 7 dscal-strength reduction, induction variable elimination and peephole optimization.

```
L5:
      mul3.d  S,vr2@<8,vr2@(0),L5      ; *(dx) *= da; dx += 8; next
```

Fig. 8 dscal-loop control optimization.

and last element to be accessed. Of course, this requires the index variable to monotonically increase or decrease by a constant value. Often this can save one instruction; in the example of dscal, which has a very tight loop of two instructions, this means a saving of 50 %.

Figure 8 shows a one instruction loop to implement the dscal routine. The instruction performs all operations : 1) scale array element, 2) increment array address, 3) compare address, and 4) branch if not all elements done.

3.7 Compiler

The GNU C compiler, which is a highly portable, full ANSI C compiler with advanced optimization features, was used as base to implement the features described in this section. To implement the dynamic range checking features, we added the following features:

- The compiler front-end generates intermediate code for dynamic range checking. The generation of intermediate for dynamic range checking may be turned on or off using a compiler switch.
- The intermediate code supports two new data types: 1) a pointer type (address) and 2) a range, a composite type consisting of two addresses (base-limit address pair).
- The GNU C compiler supports the optimization features as described in section three, but we had to extend them to support optimization of the range type. Also, register allocation had to be enhanced to support the allocation of register pairs. Various modifications found their way back into the original GNU C compiler.
- The loop control optimization is specific to the BL-register addressing scheme and required an additional optimization pass.
- We implemented code generation for the full FLATS2 instruction set.

It must be noted that all modifications and extensions did not have an adverse effect on the portability of the GNU C Compiler.

4. Evaluation

In this section we will evaluate the overhead caused by adding BL-range checking. As a benchmark, we used a C version of the Linpack benchmark. This benchmark is widely available and contains a large amount of array calculations. It tends to amplify the effect of array access and should not be considered representative of the average program. We did not use any large program like TeX or gcc to evaluate the performance aspects of the dynamic range checking code. The ratio of array access code/non-array access code is rather small in these types of programs. Furthermore, these types of programs tend to implement array access code using pointers, making the array boundary information unavailable to the compiler. However, one of the first programs to compile and execute was the gcc compiler. At that time we found a coding error which resulted in an access to an array outside its boundary (an exception was raised). This error would not have been found, if it wasn't for the range checking code.

In **Fig. 9** we give the cycle count of the inner loop of the resulting code for various BLAS (Basic Linear Algegra Subroutines) routines. The results are given for non-optimized code, optimized code, non-optimized code with range checking, optimized code with range checking, and special optimized code using the range checking hardware for loop control. The numbers given represent the cycle count of the inner loop per one iteration. The initial instructions executed are often larger for the optimized routines, as computations are moved outside the loop.

†address tag set

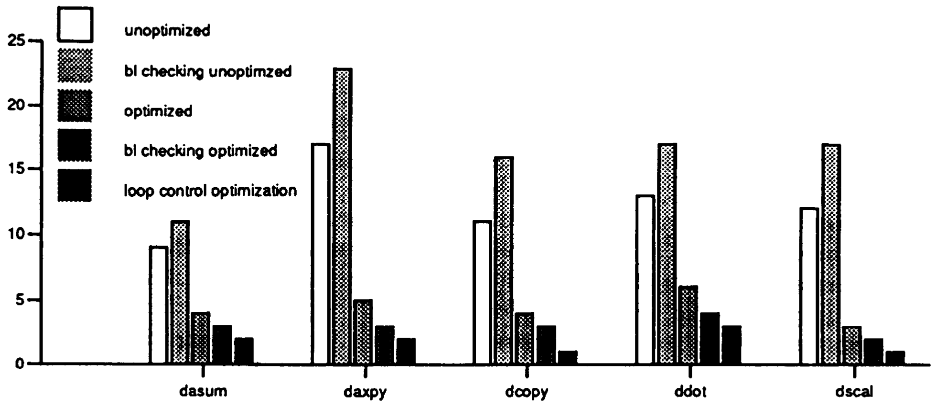


Fig. 9 Inner loop cycle count of BLAS routines.

Table 2 Result of Linpack benchmark.

	daxpy*	daxpy/daxpy$_{opt}$
unoptimized	124595614	2.76
optimized	45177340	1.00
bl checking unoptimized	177362614	3.92
bl checking optimized	28525328	0.63
loop control optimization	19998524	0.44
	total	total/total$_{opt}$
unoptimized	140639975	2.29
optimized	61220767	1.00
bl checking unoptimized	193406872	3.15
bl checking optimized	44568301	0.73
loop control optimization	36041473	0.59
	MFLOPS**	MFLOPS/MFLOPS$_{opt}$
unoptimized	0.435	0.33
optimized	1.310	1.00
bl checking unoptimized	0.327	0.25
bl checking optimized	1.907	1.46
loop control optimization	2.506	1.91

The inner loop code of the optimized code is longer than the inner loop code of the optimized code using BL-register checking because in the optimized code, access to memory is done using the base-limit register pair for the user's data space, requiring additional code to calculate the address of the current element.

Table 2 gives the results of the execution of the Linpack benchmark. We chose to vary the daxpy() routine; daxpy() accounts for more than half of the execution time. In all benchmarks, the other part of the Linpack program is compiled with standard optimization. The variations in execution time come from the variations in the execution time of the daxpy() routine.

The benchmark shows that when the benchmark program is compiled without using optimization the performance drops dramatically when using range checking. This is because each array access require two additional instructions. Optimization removes most of the overhead which caused the drop in performance of the unoptimized use of range checking. *Common subexpression elimination* will reduce the multiple array boundary calculations of one array to a single one. *Loop invariant code motion* will move the calculation of the BL-register pair outside the loop. Finally, *strength reduction* makes it possible to make optimal use of the FLATS2 addressing modes. By using the range checking as an end of loop test, we can reduce the main loop of daxpy() from three instructions (three cycles) to two instructions (two cycles).

Executing the Fortran Linpack benchmark ($n = 200$, double precision) on a Sparcstation II gave a result of 2.92 MFLOPS. However, turning on the range checking option resulted in a result of 0.93 MFLOPS. The range checking caused such a large overhead that the benchmark executed only with 32 % of the normal performance.

5. Conclusion

The current ANSI C definition is insufficient to allow optimizations on arrays such as vectorization and dynamic range checking. We were able to successfully implement dynamic range checking by allowing the declaration of a pointer to a variable size array. We summarize the following advantages:

- better opportunities to do code optimization on array access

* The numbers represent the number of instructions executed. Because FLATS2 has one and two cycle instructions, it does not represent the execution time.

** Actual performance of FLATS2 with a clock of 64.5 ns.

- code generation/code optimization for dynamic range checking
- improved readability of code accessing dynamic arrays

We presented a new memory addressing scheme which combines range checking and memory access called *BL-register addressing*. We implemented this memory addressing scheme in a C compiler to implement runtime range checking of arrays. Because the range checking can be done in parallel with the access of the memory, no additional overhead is incurred.

References

1) ANSI: *Draft Proposed American National Standard for Information Systems — Programming Language C,* X3J11/88-158 (Dec. 7 1988).

2) Aho, A. V., Sethi, R. and Ullman, J. D.: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Massachusetts (1986).

3) Allen, R. and Johnson, S.: Compiling C for Vectorization, Parallelization, and Inline Expansion, *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* Atlanta, Georgia, pp. 241-249 (June 22-24 1988).

4) Auslander, M. and Hopkins, M.: An Overview of the PL. 8 Compiler, *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, Boston, Massachusetts, pp. 22-31 (June 23-25 1982).

5) Berstis, V.: Security and Protection of Data in the IBM System/38, *Proceedings of the 8th Annual International Symposimum on Computer Architecture*, pp. 245-252 (1980).

6) Bishop, J. M.: Effective Machine Descriptors for Ada, *Proceedings of the ACM-Sigplan Symposium on the Ada Programming Language,* pp. 235-242 (Nov. 1980).

7) Buckle, J. K.: *The ICL 2900 Series*, MacMillan Press (1978).

8) DEC: *VAX Architecture Handbook*, Digital Equipment Corporation, Maynard (1981).

9) Ditzel, D. R. and McLellan, H. R.: Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero, *Proceedings of the 14th Annual International Symposium on Computer Architecture*, Pittsburgh, Pennsylvania, pp. 2-9 (June 5-8 1987).

10) Gehringer, E. F.: *Capability Architectures and Small Objects*, UMI Research Press (1981).

11) Gehringer, E. F. and Keedy, J. L.: Tagged Architectures: How Compelling Are Its Advantages, *Proceedings of the 12th Annual International Symposium on Computer Architecture*, Boston, Massachusetts, pp. 162-170 (June 17-19 1985).

12) Hill, D. D.: A Hardware Mechanism for Supporting Range Checks, *SIGARCH Computer Architecture News*, Vol. 9, No. 4, pp. 15-21 (1981).

13) Hoare, C. A. R.: Data Reliability, *ACM SIGPLAN Notices*, Vol. 10, No. 6, pp. 528-533 (1975).

14) Ichikawa, S.: A Study on the Cyclic Pipeline Computer: FLATS2, Master's thesis, Tokyo University (1987).

15) Kernighan, B. W. and Ritchie, D. M.: *The C Programming Language*, Second ed., Prentice-Hall, Englewood Cliffs, NJ (1988).

16) Markstein, V., Cocke, J. and Markstein, P.: Optimization of Range Checking, *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, Boston, Massachusetts, pp. 114-119 (June 23-25 1982).

17) Motorola: MC86020 32-Bit Microprocessor User's Manual, Second ed., Prentice-Hall, Englewood Cliffs, NJ (1985).

18) Myers, G. J.: *Advances in Computer Architecture*, John Wiley & Sons, New York (1982).

19) Organick, E. L.: *Computer Systems Organization—The B5700/B6700 Series*, Academic Press, New York (1973).

20) Radin, G.: The 801 Minicomputer, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 39-47 (Mar. 1982).

21) Shimizu, K., Goto, E. and Ichikawa, S.: CPC (Cyclic Pipeline Computer) — An Architecture Suited for Josephson and Pipelined Machines, *IEEE Trans. Comput.*, pp. 825-832 (June 1989).

22) Wikes, M. V.: Hardware Support for Memory Protection: Capability Implementations, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS*), Palo Alto, California, pp. 107-116 (Mar. 1-3 1982).

Appendix A. FLATS2 Hardware

This appendix describes the FLATS2 hardware features not described in the main paper, including the functional diagram and the FLATS2 pipeline stages. The FLATS2 was developed to provide support for both symbolic and numerical applications.

FLATS2 functional units

Figure A1 shows the functional diagram of the FLATS2. The Instruction Processing Unit (IPU) fetches the instruction from the instruction memory (IM) and decodes the instruction. It will also calculate the branch target when necessary. The Global/Virtual frame Unit handles access to the global and local frame registers, handles simple integer arithmetic, supports compare and branch and Base-Limit register checking. The Sum and Product Unit provides integer and floating point operations including special instruction such as a real inner product (rip) instruction.

FLATS2 pipeline

Figure A2 shows the FLATS2 pipeline stages. During the first stage (IF), the instruction is fetched. The instruction is then decoded during the second stage (ID). Registers required by the instruction are fetched during the third stage (GVR). During the fourth stage (GVX) the effective address is calculated and compared against the Base and Limit values. It also handles simple integer arithmetic operations on registers. This stage also implements the add-compare-branch instruction group. The next stage (DMR) reads the data required by the operation from data memory. During the sixth to ninth stage (EX1-EX4) integer and floating point operations take place. Also, during the sixth stage (GRW) a register write takes place, writing the result of a simple integer arithmetic operation or the result of the use of an addressing mode with side effect. In the tenth and final stage (DMW) the result of an operation is written to data memory.

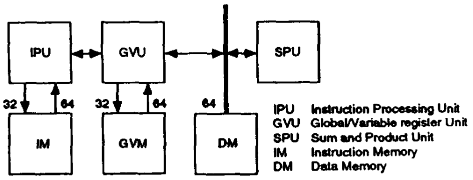


Fig. A1 FLATS2 functional diagram.

FLATS2 implementation

The FLATS2 was implemented using 10 K and 100 K ECL technology and has a machine cycle time of 65 ns. Physical memory is 5.5 MB. A total of 27 boards measuring 47 × 32 cm were required for the implementation.

FLATS2 addressing modes

The FLATS2 has three basic addressing modes (**Table A1**). In index mode, the contents of a register (contains non-address value) is added to the base address to form the effective address. In offset mode, an immediate constant is added to the base address to form the effective address. And in pointer mode, the effective address if formed by an immediate displacement to the contents of a register containing an address. Each addressing mode can include either a pre/post increment/decrement side-effect.

FLATS2 instruction format

The FLATS2 has various instruction formats. The M-format is the format for arithmetic operations (**Fig. A3**). Each instruction includes a branch field j0 which specifies the branch offset

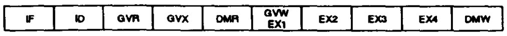


Fig. A2 FLATS2 pipeline stages.

Table A1 FLATS 2 addressing modes.

Mode	Notation	Effective address	Side-effect
Index	*base @ index*	BL : *base + index*	
Index push	*base @ > index*	BL : *base + index*	*base* ← *base + index*
Index pop	*base @ < index*	BL : *base*	*base* ← *base + index*
Offset	*base @ displ*	BL : *base + displ*	
Offset push	*base @ > displ*	BL : *base + displ*	*base* ← *base + displ*
Offset pop	*base @ < displ*	BL : *base*	*base* ← *base + displ*
Pointer	*base* : *displ* (*pointer*)	BL : *pointer + displ*	
	base : & *address* (*index*)	BL : *address + index*	
Pointer push	*base* : > *displ* (*pointer*)	BL : *pointer + displ*	*pointer* ← *pointer + displ*
Pointer pop	*base* : < *displ* (*pointer*)	BL : *pointer*	*pointer* ← *pointer + displ*
Index offset	*base @ displ* (*index*)	BL : *base + index + displ*	
Pointer index	*base* : *displ* (*pointer*) *index*	BL : *pointer + displ + index*	

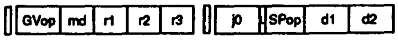


Fig. A3 FLATS2 M-format instruction format.

of the branch to be taken on a successful memory access. If an access error occurs, that is, the either the effective address or the base and limit addresses are not a valid address or the effective address falls outside the address range specified by the base and limit addresses, the instruction execution is aborted and the next instruction is executed. If the branch offset specifies the next instruction, the next instruction is executed when no access error occurs. If an access error occurs, an exception is raised.

(Received May 29, 1991)
(Accepted October 21, 1992)

Paul Spee (Member)

Paul Spee was born on April 15, 1960 in Dordrecht, the Netherlands. He received his M. S. degree from the Delft University of Technology in 1986. From 1984 to 1986 he was employed by the Delft University of Technology. From 1986 to 1991, he was a researcher with the Quantum Magneto Flux Logic Project (ERATO). In 1991, he joined Unix System Laboratories, Pacific as a technical manager. His areas of interest include parallel architectures, operating systems, and parallel and concurrent programming languages.

Eiichi Goto (Member)

Eiichi Goto was appointed Assistant Professor, Associate Professor and Professor of the Faculty of Science, University of Tokyo in 1958, 1959 and 1970, respectively. He was appointed Chief Researcher of RIKEN in 1968, Director of the Quantum Magneto Flux Logic Project of JRDC in 1986 and Director of the Computer Center at the University of Tokyo from 1987 to 1991. His main interests have been in the field of Parametron Computer, Magnetic Monopole, Electron Beam Exposure System, Flux Transfer Josephson Device (Quantum Flux Parametron) and High Performance Supercomputer Architecture. He has been awarded many prestigious prizes including a commendation by the Minister of State of Science and Technology for his research on the Parametron in 1959, the Okochi Memorial Technology Prize in 1988 and the Purple Ribbon Medal in 1989 for his research on the Variable Shape Electron Beam Exposure System.