Implicit Array Bounds Checking on 64-bit Architectures

CHRIS BENTLEY, SCOTT A. WATTERSON, DAVID K. LOWENTHAL, and BARRY ROUNTREE
The University of Georgia

Several programming languages guarantee that array subscripts are checked to ensure they are within the bounds of the array. While this guarantee improves the correctness and security of array-based code, it adds overhead to array references. This has been an obstacle to using higher-level languages, such as Java, for high-performance parallel computing, where the language specification requires that all array accesses must be checked to ensure they are within bounds. This is because, in practice, array-bounds checking in scientific applications may increase execution time by more than a factor of 2. Previous research has explored optimizations to statically eliminate bounds checks, but the dynamic nature of many scientific codes makes this difficult or impossible. Our approach is, instead, to create a compiler and operating system infrastructure that does not generate explicit bounds checks. It instead places arrays inside of *Index Confinement Regions* (ICRs), which are large, isolated, mostly unmapped virtual memory regions. Any array reference outside of its bounds will cause a protection violation; this provides *implicit* bounds checking. Our results show that when applying this infrastructure to high-performance computing programs written in Java, the overhead of bounds checking relative to a program with no bounds checks is reduced from an average of 63% to an average of 9%.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Optimization*

General Terms: Measurement, Performance

Additional Key Words and Phrases: Array-bounds checking, virtual memory, 64-bit architectures

1. INTRODUCTION

One of the long standing issues in programming languages and compilers concerns checking array bounds to ensure program correctness. The simplest solution is for the compiler to generate bounds-checking code for each array reference. If the reference is outside the bounds of the array, a run-time error is generated. Unfortunately, this simple solution adds overhead to all run-time array accesses. Therefore, languages focused on efficiency, such as C, do not

Authors' address: Chris Bentley, Scott A. Watterson, David K. Lowenthal, and Barry Rountree, Department of Computer Science, The University of Georgia, Georgia; email: {cbentley,saw,dkl, rountree} @cs.uga.edu.

require checking of array bounds. Despite their overhead, array-bounds checks are important because array-access violations are a frequent source of error. They are particularly difficult to detect and fix because these violations may cause errors in seemingly unrelated parts of the code. For this reason, some languages (e.g., Java and C#) require array-bounds checking.

Meanwhile, interest is growing in the scientific programming community in using higher-level languages, such as Java, for high-performance computing (HPC). Some reasons for this include attractive features, such as portability, expressiveness, built-in threads, and safety. However, several obstacles currently prevent high-level languages from becoming widely accepted for HPC—one of which is the array-bound checking requirement, which is common in higher-level languages. While array accesses are infrequent in some applications, scientific programs tend to be array intensive, which means that checking array bounds can significantly increase execution time. For high-level languages to have any chance to be widely used for HPC applications, array-bounds checking overhead must be alleviated.

The two dominant practices for reducing bounds-checking overhead are static and dynamic analysis. The former, which is most appropriate for a traditional language like C, attempts to eliminate explicit checks by proving an array reference is within its bounds [Bodik et al. 2000]. However, there are several problems with this approach. First, the dynamic nature of many scientific codes makes static elimination difficult or impossible. This is borne out by our manual inspection of the NAS benchmark suite [Bailey et al. 1991]. Second, even when static analysis is possible, currently available compilers may have difficulty removing explicit checks. Dynamic analysis, on the other hand, is most applicable to modern (dynamic) languages, such as Java. It strives to analyze the code at run-time to find regions where array-bounds checks can be eliminated and then generates a new customized routine for that region. Examples of this approach include Java HotSpot [SUN 2002] and JRockit [BEA 2005].

Rather than attempting to eliminate explicit bounds checks statically, our goal is to use compiler and operating system support to implicitly check array bounds. We leverage the 64-bit address space of modern architectures to reduce the cost of array-bounds checks. This is a potentially useful technique for scientific applications, which are increasingly difficult to analyze statically. If static analysis cannot eliminate bounds checks, then the compiler must insert up to $2n$ bounds checks for an n-dimensional array reference.

Instead, we perform no static analysis of the program and we insert, at most, one check per array reference. We do this by placing each array object in an *index confinement region* (ICR), which is an isolated virtual memory region, of which only a small portion, corresponding to valid array data, is mapped and permissible to access. The rest of the ICR is unmapped and any access to that portion will cause a hardware protection fault. This achieves implicit bounds checking for all array references. While ICRs can be implemented on most modern architecture/operating systems, they are primarily intended for 64-bit machines. This allows allocation of hundreds of thousands of 32 GB ICRs, each of which is large enough to implicitly catch any illegal access to an array of double-precision numbers.

Our initial implementation was within Java, where we made two primary modifications to `gcj`, the GNU Java implementation. First, we modified array allocation (`new`) so that array objects are allocated inside of ICRs. Second, we modified the way `gcj` generates array-indexing code so that illegal accesses will fall into an unmapped memory area. Our technique fully supports multithreaded Java programs.

Because our new Java implementation utilizes ICRs, dense access patterns are replaced with sparse ones. This has negative consequences in the TLB, cache, and memory. Accordingly, we modified the Linux kernel to customize virtual addressing to optimize for a sparse access pattern. Furthermore, a process can selectively choose to use our customized scheme, so that regular processes are unaffected by our kernel modifications.

Our results on a 900-MHz Itanium-2 show that performance of our new Java implementation that uses ICRs is superior to `gcj`-generated code that uses explicit bounds checks. Specifically, our approach reduces the bounds-checking overhead (relative to performing no checks) for scientific Java benchmarks from an average of 63% to an average of only 9%. In addition, *all* of the benchmarks performed better using ICRs, rather than full compiler bounds checking.

The remainder of this paper is organized as follows. The next section describes related work; Section 3 shows that array-bounds checking incurs significant overhead on a variety of architectures. Next, Section 4 provides implementation details and Section 5 presents performance results. In Section 6, we show the generality of the ICR technique by describing an implementation specific to the C language. Section 7 discusses issues arising in this work. Finally, Section 8 concludes.

2. RELATED WORK

A significant body of work exists on static analysis to eliminate array-bounds checks. Midkiff et al. [1998] finds *safe regions* that decrease the impact of Java's required run-time checks as well as its precise exception model and a follow-on paper described a new array class for Java that provides true multidimensional arrays [Moreira et al. 2000]. The technique in Midkiff et al. [1998] is applied automatically within a compiler Artigas et al. [2000]. It targets scientific applications written in Java. Kolte and Wolfe [1995] perform partial redundancy analysis to hoist array-bound checks outside of loops. Their algorithm was based on that described by Gupta [1993], who formulated the problem as a data-flow analysis. In ABCD, Bodik et al. [2000], implemented a demand-driven approach to bounds checking in Java. Xi and Pfenning [1998] introduce the notion of dependent types to remove array-bound checks in ML; Xi and Xia [1999] extend this idea to Java. Rugina and Rinard [2000] provide a new framework using inequality constraints for dealing with pointers and array indices, which works for both statically and dynamically allocated areas. Early work was done by Markstein et al. [1982] on statically removing bound checks.

Most of the above analyses are performed purely at compile time. This has the advantage of avoiding run-time overhead, but fails when either (1) the code

Table I. Percentage of Bounds Checks[a]

Program	Dynamic Percentage of Removable Checks
FT	0%
MG	90%
BT	100%
CG	7%

[a] Bounds check for four NAS programs, that we believe could be eliminated statically. In practice, we are not aware of a compiler that can successfully eliminate *all* of the removable checks indicated above.

is too complicated to prove anything about array references or (2) the code depends on input data. The former includes cases where, for example, different arrays are passed (as actual parameters) to functions from several different call sites. The latter includes applications where indices cannot be determined at compile time. As one example, Table I shows the results of our inspection of several benchmarks from the NAS suite. In particular, our inspection of the CG and FT benchmarks show that it is extremely difficult and impractical to prove that array references are within bounds. Furthermore, even when most or all checks are *potentially* removable, we are not aware of a compiler that will successfully eliminate *all* removable checks. In these cases, compile-time schemes must fall back to general run-time checking. Instead, our implementation decreases the cost of bounds checking and does not depend on static analysis to do so.

A similar approach to ours is to use segmentation for no-cost bounds checking [Lam and Chiueh 2005]. The basic idea is to place an array in a segment and set the segment limit to the size of the array. This technique is effective for small one-dimensional arrays, because automatic checking is done on both ends of the array. However, typically, one end must be explicitly checked for large arrays. Furthermore, because there are a limited number of segments that can be simultaneously active (four on the x86, for example), full bounds checking must be used for some arrays if there are more live arrays than this maximum. Most importantly, multidimensional arrays cannot be supported. This is because the segment limit prevents only an access past the allocated memory for the entire array; an illegal access in one of the first $n - 1$ dimensions that happens to fall within the allocated memory for the array will not be caught. While this provides some degree of security, it can not produce semantically correct Java programs.

Electric Fence [Perens], which places a single unmapped page on either side of an allocated memory area, bears some similarity to ICRs . Electric Fence automatically catches overruns on one end. However, it does not handle arbitrary array references, such as references past the unmapped page. In contrast, our Java implementation is able to catch any illegal reference.

Our approach bears some similarities to Millipede [Itzkovitz and Schuster 1999], a software DSM system. Millipede avoids thrashing by placing distinct variables (that would generally be allocated on the same page) on different pages at their appropriate offsets; both pages are then mapped to the same physical page. Different protections can then be used on each variable, because protections are done at the virtual page level.

Table II. Pentium Assembly Code Produced by `gcj` Both Without (left) and With (right) Bounds Checking[a]

Instruction	Description
movl (A), %ecx	get A
movl $999, 28(%ecx)	A[5] = 999
ret	return

Instruction	Description
movl (A), %ecx	get A
cmpl $5, 4(%ecx)	compare to high bound
jae L3	jump to trap if > bound
movl $999, 28(%ecx)	A[5] = 999
ret	return
L3: pushl $5	push offending index
call ThrowBadArrayIndex	generate exception

[a]Some low-level details are elided for clarity.

Our extended and customizable virtual memory abstraction (*xvm*) is used only by those processes that use ICRs . This means that regular processes are unaffected. This is somewhat reminiscent of microkernels, such as Mach [Young et al. 1987], which provide flexibility to application-level processes (such as allowing an external pager). However, our modification is inside the kernel as opposed to at the application level.

There has been work on how to utilize 64-bit architectures, mostly from the viewpoint of protection. For example, Chase et al. [1994] describes Opal, which places all processes in a single 64-bit virtual address space. This allows for a more flexible protection structure.

We should note that there exists research that claims that bounds checking on modern processors is only modest. For example, Boisvert et al. [1998] found a 20% overhead on one scientific program. However, our work, as well as that of Lam [Lam and Chiueh 2005], shows a *much* larger overhead.

Finally, array-bounds checking is often mentioned as a technique for preventing buffer overflow. Several have studied the general overflow problem; this includes using a `gcc` patch along with a *canary* to detect it [Cowan et al. 1998]. Another compile-time solution, RAD [Chiueh and Hsu 2001], involves modifying the compiler to store return addresses in a safe location. This solution retains binary compatibility because stack frames are not modified.

3. ARRAY-BOUNDS CHECKING OVERHEAD

While this paper focuses on an Itanium-2 platform, bounds checking causes significant overhead on a range of architectures. The previous section showed that static analysis cannot always allow for the removal of bounds checks. This section shows that if the checks cannot be removed, there is inherent overhead on both the Itanium architecture *and* the much more ubiquitous Pentium architecture. (Note that here we assume that *only* the high-bound check must be made; Section 4 shows how it is that the low-bound check can be omitted in the specific case of Java. In C, both low- and high-bound checks must be made.)

Figure 1 shows a single array access; Table II shows the `gcj` (with the highest optimization level)-generated code with and without bounds checking on

```
void foo() {
  A[5] = 999;
}
```

Fig. 1. Example source code.

Table III. Itanium Assembly Code Produced by `gcj` (Without Bounds Checking)[a]

Instruction 1	Instruction 2	Instruction 3	Description
nop	addl r15 = 999, r0	r3 = A	set r15 to 999, get address of A
ld8 r2 = [r3]	adds r14 = 32, r2	nop	load and add to get address of A[5]
nop	st4 [r14] = r15	nop	store into A[5]
nop	nop	ret	return

[a]Some low-level details are elided for clarity.

Table IV. Itanium Assembly Code Produced by `gcj` (With Bounds Checking)[a]

Instruction 1	Instruction 2	Instruction 3	Description
r8 = A	nop	nop	get address of A
ld8 r3 = [r8]	adds r15 = 8, r3	adds r16 = 32, r3	load base of A, get addresses of high bound and A[5]
ld4 r2 = [r15]	cmp4.ltu p6, p7 = 5, r2	nop	get high bound, load high bound
(p7) addl r35 = 5, r0	(p7) call ThrowBadArrayIndex	nop	if violation, store offending index and throw exception
addl r14 = 999, r0	st4 [r16] = r14	nop	set r14 to 999, store into A[5]
nop	nop	ret	return

[a]Some low-level details are elided for clarity.

the Pentium 4.[1] In the bounds-checking version of the code, two additional instructions are executed. The bounds check also involves an additional memory reference because of the `cmp` instruction.

This is the minimal amount of code that can be executed if program correctness is to be maintained. Clearly, there must be a comparison and conditional jump. Hence, at least two additional instructions must be executed for a bounds check in Java.

Table III and IV show the generated code without and with bounds checking on the Itanium. This code is more complicated to understand because of the *bundles* present on the VLIW-based Itanium. Each (long) instruction contains three regular instructions and dependences (such as read after write) can appear in a bundle as long as there is a *stop* placed in between instructions (not shown in the figure) [Intel 2006]. In addition, any of the three instructions in a bundle may be *predicated*. Such an instruction will only be executed if a predicate register is set to true. These predicate registers are set by `cmp`

[1]With `gcj`, metadata are stored in front of actual array data. The convention is that a pointer to the object descriptor is stored first (which is 4 bytes on the Pentium and 8 bytes on the Itanium), and the array length is stored after that (4 bytes on both architectures). The first data element of an array is located directly after the length.

instructions. In the example code shown in Table IV, the array-bound checks are implemented with such a sequence (using predicate $p7$).

On the Itanium, the bounds check incurs a load, a compare, and a branch instruction. In the example shown in Table III, there are `nop` slots in some bundles. However, as can be seen in Table IV, scheduling constraints make it impossible to insert all bounds-checking instructions into `nop` slots. In the example shown in Table IV, the bounds-checking version executes two additional bundles. While it is conceivable that in this example five (instead of six) bundles could be used, dependences would eliminate advantages of such packing. The key point is that there is no way to get around a strict (significant) increase in the total number of bundles.

Overall, bounds-checking instructions that cannot be removed statically will cause a significant amount of overhead for the particular load or store that is being executed—and this overhead is not specific to the Itanium. In scientific programs, which are often array intensive, this translates to a significant program overhead.

4. IMPLEMENTATION

This section describes implementation details on an Itanium-2, which is a 64-bit machine. First, we discuss our implementation of index-confinement regions (ICRs). Next, we discuss our modifications to the GNU Java implementation, `gcj`. Finally, we describe our modifications to the IA-64 Linux kernel.

4.1 Index-Confinement Regions

An Index-Confinement Region (ICR) is a large, isolated region of virtual memory. When an array is placed in an appropriately sized ICR, references to this array are confined within the ICR. For example, consider placing a one-dimensional integer array, indexed only by 32-bit expressions, within an ICR. If the size of the ICR is chosen to be at least 16 GB and the array is placed at the beginning of the ICR, it is impossible to generate an array reference that is outside of the ICR. A reference below the lower end of the ICR is not possible because of three factors. First, arrays in Java are limited to 2^{31} entries by the language specification. Second, negative-index expressions are not permitted by the Java language specification. Third, our Java compiler (as well as others) takes advantage of these language restrictions by treating index expressions as unsigned, so that if a negative 32-bit index expression is generated by a program, it is converted to a positive index expression larger than $2^{31} - 1$. Note that this simple optimization cannot be performed in the general case by a C or C++ compiler, because negative index expressions are legal in those languages.

An ICR must be large enough so that any 32-bit index expression will result in an access within an ICR. In general, ICR size is the product of 4 GB (2^{32}) and the size of the array element type; this can be calculated at allocation time. Although each ICR is several gigabytes in size, a 64-bit virtual address space permits the allocation of millions of ICRs.

Figure 2 shows two regions and their associated arrays. In an ICR, pages in which the actual array resides are mapped with read and write permission.

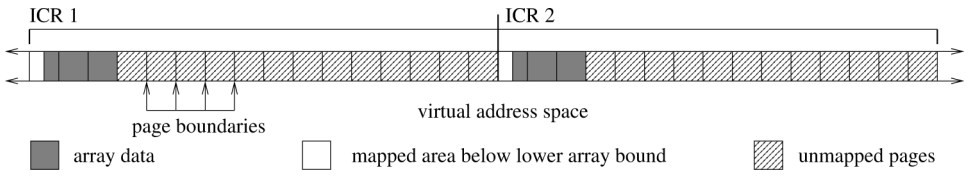

Fig. 2. Two index-confinement regions. Each array is isolated from any other program data; bounds checking is done automatically through unmapped pages.

All other pages in the ICR are unmapped. This is achieved using `mmap` (with `MAP_FIXED`). Because, in general, the array size is not a multiple of the page size, one end of the array that is within a mapped page is not implicitly protected. We align arrays so that the upper limit of the array is at a page boundary, leaving the bottom end of the array unaligned.[2] This allows automatic bounds checking, because an access to an unmapped page results in a protection violation. Leaving the front end of the array unprotected (not page aligned) does not matter because of the treatment of negative indices, described above. Finally, we can suppress null pointer checks and customize the protection violation code to determine whether the violation was as a result of a null pointer or to a reference beyond the end of an array.

The ICR abstraction extends naturally to n dimensions, because Java has no true multidimensional arrays. Instead, Java represents an n-dimensional array as vectors of vectors. As described above, Java compilers already can avoid a lower-bound check for each vector. Hence, because each vector is placed in an ICR, the high-bound check is also unnecessary and *no* checks are required for an n-dimensional array reference.

Because Java requires arrays to be allocated via the keyword `new`, the run-time system must be modified to place arrays in ICRs. In addition, the Java compiler must be modified so that implicit bounds checking can be performed. The next section describes how we modified the `gcj` compiler and run-time libraries to facilitate the ICR-based allocation of Java arrays.

4.2 GNU Java Implementation

We created a new Java implementation, based on `gcj`, that makes use of ICRs. We made two primary modifications. First, we changed the way arrays are indexed and allocated, which involves both compiler and run-time library changes. Second, we changed the GNU backend so that it would conform to the Java language specification.

4.2.1 *Array Indexing and Allocation.* Java requires that array-index expressions produce a positive offset that is within the allocated space of the array. This means that conceptually `gcj` must produce two checks per access—one to check that the index is positive and one to verify the index is less than the length of the array. The length of each array is stored as a field in the array object. However, as mentioned above, `gcj` optimizes these checks into a single

[2]Because we use right-aligned placement of arrays, ICRs require one extra page at the right end to ensure proper protection.

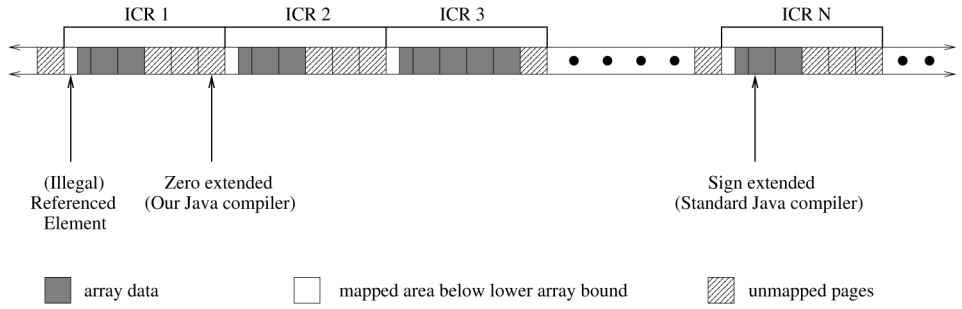

Fig. 3. Pictorial description of an illegal array reference and its corresponding sign-extended index expression versus zero-extended index expression with ICRs .

check by sign-extending indices and comparing the index to the array length as an unsigned value. Therefore, any negative index becomes a very large positive index that is guaranteed to be larger than the length of the array. This is as a result of the restriction that the number of array elements in a Java array is limited to the maximum signed integer ($2^{31} - 1$).

We modified `gcj` to target ICRs . First, we had to disable bounds checking in the compiler, which was easily done as `gcj` provides a compile-time flag for this purpose. Second, because the precise location of ICRs is not known at compile time, we cannot simply sign-extend the index expression, as this might result in an access to a mapped portion of a different ICR when added to the array base address (see Figure 3). While this is not a problem in standard `gcj` , because an explicit comparison is made to the upper bound, our implementation makes no comparisons. As a result, we must map a negative index expression to a page that is guaranteed to be unmapped. Therefore, instead of sign-extending the index expression, we zero-extend it. This ensures that any negative expression becomes an unsigned expression precisely in the range of index expressions (2^{31}, $2^{32} - 1$). Such an indexing expression will result in an access to an unmapped page within the ICR *for that array*.

In Java, all arrays are allocated dynamically with the keyword `new` , which calls a run-time library routine appropriate for the array type (object or primitive). These particular routines are `NewObjectArray` and `NewPrimArray`, respectively. Each of these functions eventually calls `malloc` to actually allocate the array. When using ICRs , we modified both of these routines, replacing `malloc` with a call to `mmap` with (1) the target address of the next available ICR and the (2) the `MAP_FIXED` flag (as described in the previous section). These methods extend naturally to n-dimensional arrays, as the first $n - 1$ dimensions are arrays of objects. The keyword `new` invokes `NewMultiArray`, which invokes itself recursively, calling `NewObjectArray` until all of the first $n - 1$ dimensions are allocated. Finally, the last dimension will be allocated using `NewPrimArray` if the type is primitive; otherwise, it is allocated using `NewObjectArray`.

Because all ICR support is implemented in the compiler, run-time libraries, and operating system, no source code modification is necessary to allocate arrays in ICRs . However, allocating arrays in ICRs poses many challenges to the

operating system and architecture. Section 4.3 discusses the impact of ICRs on the memory hierarchy along with OS support for ICRs .

4.2.2 *GNU Backend Modifications.* As described above, our ICR technique allows `gcj` to avoid generating array-bound checks into the intermediate code. Potentially, this could enable aggressive optimizations, specifically instruction scheduling, that could result in a violation of the Java semantics. The particular situation we face is that without bounds checks, the optimizer might reorder array references, which is a potentially unsafe optimization. (This is because we catch out-of-bounds via an OS exception, but the compiler is not aware of this.)

To prevent this problem, we modified the GNU backend so that it would not reorder array references. This was done in two steps. First, the syntax tree created by `gcj` is inspected and, whenever an array reference tree node is found, a bit in its RTL representation is set. Second, the instruction scheduler scans for this bit and moves the RTL corresponding to the array reference earlier in the generated code only if it is not reordered with another array reference. We note that this does not prevent reordering of scalar variables with array references. Our prototype could be modified to prevent this, but we are targeting scientific applications (our tests use mostly the NAS suite). This situation is uncommon—most scalars are actually compiler-generated temporaries whose state is hidden from the programmer.

4.3 Linux Support for ICRs

While our new Java implementation allocates arrays in ICRs , obviating the need for bounds checks, the ICR abstraction itself places significant pressure on the memory hierarchy—causing cache conflicts and internal fragmentation. Smaller page sizes lessen this problem, although the address space available in Linux (and, hence, the number of ICRs that can be used) decreases with the page size because of the three-level linear page table in Linux. A more subtle problem is that ICRs force a sparse access pattern, which causes the kernel to consume significant amounts of memory to hold page tables.

To mitigate these problems, we have designed and implemented an abstraction we call *xvm* to provide an application process with an extended, customizable virtual memory. As we are interested in ICR -based programs, we use the extended virtual address space to allocate as many ICRs as are needed and a customized virtual addressing scheme to reduce memory consumed by page tables.

The first part of implementing an *xvm* is to increase the address space size. Linux uses a three-level page table (PT) for translation. Borrowing Linux terminology, we refer to a page table as a *directory*. Thus, the first, second, and third levels of the PT are denoted L1PD, L2PD, and L3PD. The L3PD contains entries that map the virtual page to a physical frame. In standard IA-64 Linux (see Figure 4, largely borrowed from Mosberger and Eranian [2002]), a 4 KB page size provides an 320-GB address space, which is far too small for any of our benchmarks, while a 64-KB page size provides 20 PB of address space, which is sufficient for all of our benchmarks. We modified the 4-KB kernel directory

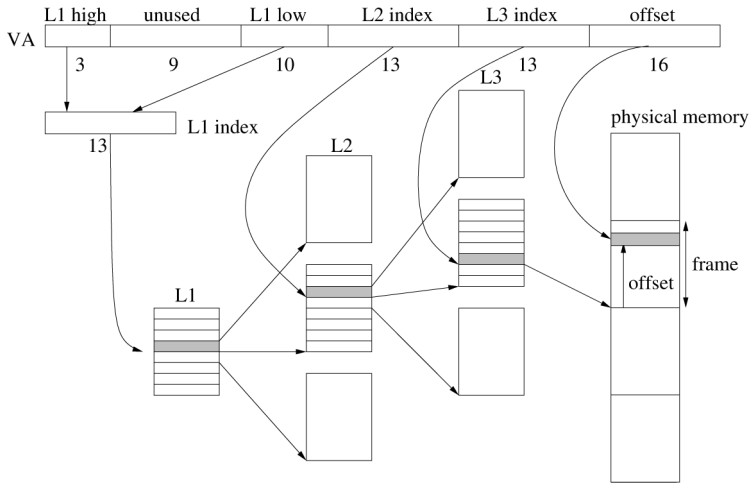

Fig. 4. Address space layout and address translation for regular processes with a three-level page table in Linux, using 4 KB pages.

structure to allow the large virtual address space allowed by the 64-KB kernel, while maintaining for ICRs the cache and memory benefits of the 4-KB page size. Simply stated, we modified the L2PD and L3PD to be held in several consecutive pages. This means that the L2PD and L3PD directories each consume N consecutive pages and need $log_2 N$ additional bits for their index. For example, increasing the L2PD and L3PD to 512 pages each results in an available virtual address space of over 32 PB, which is adequate space for hundreds of thousands of ICRs . Figure 5 shows the new scheme. The total number of bits that we use in a virtual address is the (Itanium) hardware limit of 57, which allows allocation of hundreds of thousands of regions. The mid- and lower-level directories are both 18 bits wide. This allows for the minimum initial allocation overhead among all possible partitionings of 36 bits. Furthermore, the maximum additional memory usage due to the L2PD is 128 MB, which only occurs if all of virtual memory is used.[3]

The second part is customizing the virtual addressing scheme. As mentioned above, the sparse access patterns imposed by ICRs violate the principle of locality. For example, a 4-KB page size directory contains 512 entries, so in an extended virtual address space as described above, each L3PD contains 256 K entries (512 pages $\times$ 512 entries per page), assumed to represent *contiguous* virtual memory. Hence, one L3PD represents 1 GB (256 KB $\times$ 4 KB) of virtual memory. Unfortunately, arrays placed in ICRs are at least 4 GB from one another. This means that when using the scheme shown in Figure 5, two different array references require allocation of *two* L3PDs—even though typically only *one* entry in each L3PD will be used. As a result, memory consumption because of directories is increased considerably. This is a well-known problem

[3]We could also increase the number of bits in the first level, which would decrease the initial allocation overhead, but would increase the number of L2PD and L3PD allocations.

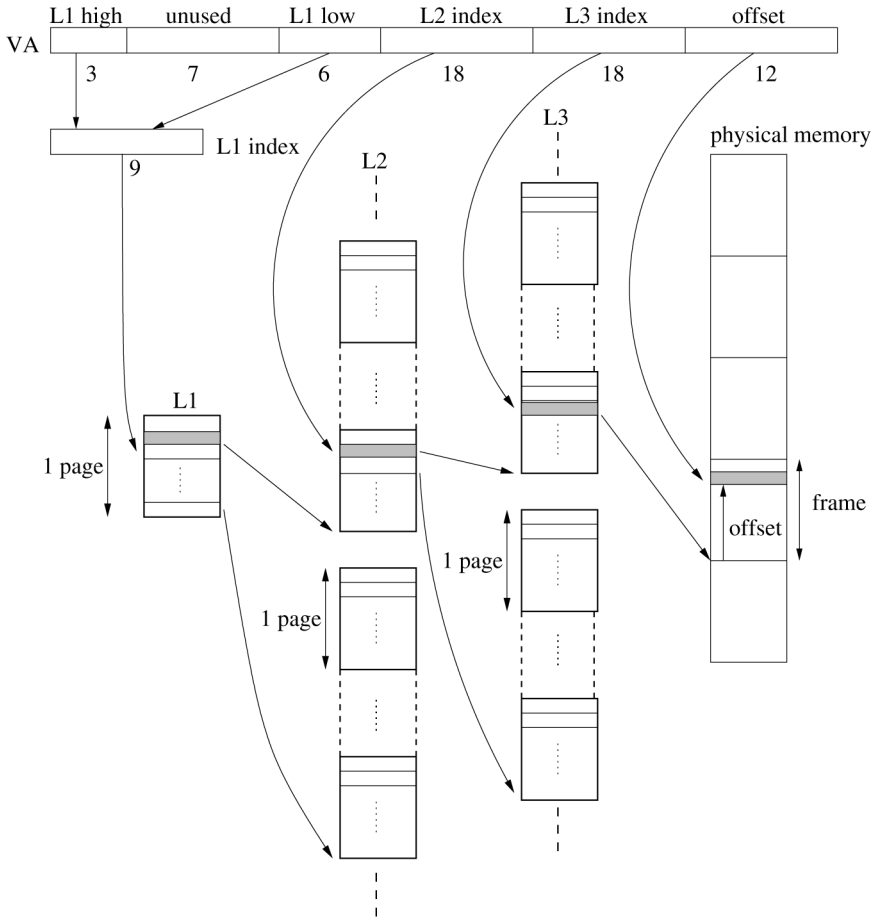

Fig. 5. Address space layout for processes using our *xvm* abstraction. The L2PD and L3PD are each 512 instead of 1 page.

with extremely sparse address spaces, which are generally better served using a hashed or clustered page table [Talluri et al. 1995].

However, ICRs exhibit regular sparse patterns; this is especially true for multidimensional arrays, where a given dimension has vectors with identical sizes and types. We take advantage of this by swapping bits between the L3PD and the L2PD indices, so that the L3PD can hold many ICRs (rather than one). Each page of the L3PD contains entries for consecutive pages, so ICRs that have consecutive pages mapped use consecutive entries in the L3PD (up to a limit). Our implementation significantly reduces the internal fragmentation of the directories, leading to a reduction in memory usage by processes using ICRs . The original and new page table indexing schemes are shown pictorially in Figure 6. Without our customized addressing scheme, the kernel runs out of memory as a result of the large number of directory allocations in Multigrid and Fourier Transform. With our scheme, both programs run successfully.

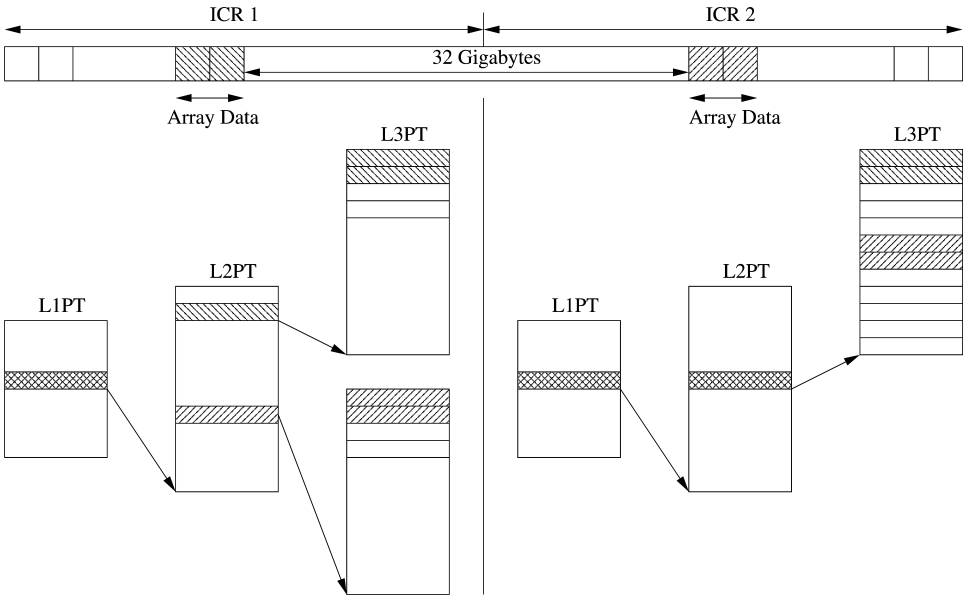

Fig. 6. The original (left) and *xvm* (right) page table-indexing schemes. The fill patterns shown in the directories correspond to the mapped pages in the ICRs.

To avoid increasing the minimum size and memory usage of processes that do *not* use ICRs, an *xvm* is used exclusively via a new system call `enable_xvm` (invoked via the Java run-time library), which converts a standard Linux three-level page table into an multipage, multilevel extended version (as shown in Figure 5). This allows only those processes that make use of ICRs to incur the memory overhead because of the larger directories when using *xvm*.

Despite a significantly lower directory memory footprint, customizing the virtual addressing scheme incurs some performance penalties. Several Linux routines operate on a range of virtual addresses; they look up only the start and end address in the page table; the routine can then process all the intervening directory entries using simple pointer arithmetic. With the modified addressing scheme, this assumption is incorrect, as consecutive entries in the L2PD and L3PD do not reference contiguous memory. As a result, each time a new directory entry is needed, a full lookup of the virtual address is required, which adds overhead. However, in practice, this overhead was negligible in our benchmarks.

Also, the VHPT walker, which performs page look-ups in hardware on the Itanium, assumes a linear or hashed page table. Our scheme is neither and, therefore, cannot use the VHPT to update the page table for ICRs. Instead, we handle page faults inside an *xvm* in such a way that after a TLB miss, the VHPT will always incur a miss. In this way processes that do not use *xvm* can still use the VHPT as normal. In practice, we found that incurring a miss in the VHPT each time in programs using ICRs had little effect on overall execution time—this is because our benchmark programs (and scientific programs in general) typically have good locality.

5. EXPERIMENTAL RESULTS

We examined the performance of both our new Java implementation, as well as standard `gcj` on a variety of Java applications. These applications include the Java version of the NAS Parallel Benchmark Suite 3.0 [Frumkin et al. 2002], some simple hand-written scientific kernels, and a synthetic array program. While the NAS programs can be executed with multiple threads, we run the serial versions because our experimental platform is a uniprocessor. We emphasize that the technique used in our new Java implementation is completely applicable to multithreaded Java programs.

Each NAS program uses Class W input size. The NAS Java programs include **CG**, a conjugate gradient program; **FT**, a fourier transform; **IS**, an integer-sorting program; **LU**, a regular-sparse, block triangular system solution; **MG** (and **MG3D**), a multigrid solver; and two computational fluid dynamics simulations, **BT** and **SP**. Other than **MG3D**, the Java versions of the NAS suite use linearized arrays rather than multidimensional ones; this will be discussed later. Our hand-written kernels use multidimensional arrays and include **MM**, matrix multiplication; **JAC**, a Jacobi iteration program; and **TOM**, the TomcatV mesh generation program from SPEC92, which we converted to Java. The input sizes for these three programs were 896×896, 896×896, and 768×768, respectively, and were chosen by finding the size that resulted in execution time with no bounds checking taking 30 s. In addition, we include three versions of a synthetic benchmark, **S1D**, **S2D**, and **S3D**, with array sizes $1M$, 1000×1000, and $100 \times 100 \times 100$, respectively. The total number of elements in each of these arrays is the same. This benchmark simply repeatedly updates each element of an array and is used to study how ICRs and bounds checking scale with dimensionality. We used maximum optimization (-O6) to compile our programs with both Java implementations. In addition, we compiled all benchmarks directly to executable programs rather than Java bytecodes.

We performed our experiments on a 900-MHz Itanium-2 with 1.5-GB memory, a 16-KB L1 instruction cache, 16-KB L1 data cache, 256-KB L2 cache, and 1.5-MB L3 cache. The operating system is Debian Linux version 2.4.19. We use wall clock times for measurement; all experiments were run when the machine was unused.

The rest of this section is organized as follows. First, we present the overall execution times of several programs. We then further examine some of the results through inspection of hardware-level counters.

5.1 Overall Execution Times

Figure 7 shows the execution time of each version of the NAS Java benchmarks. The baseline version, labeled *No Checks* , is compiled with `gcj` using a compile-time flag to disable bounds checks. All other versions of each program are normalized to this value. *Full Checks* is the default program produced by `gcj` ; all bounds are checked via compare instructions in the code. Each access to an n-dimensional array incurs a single upper bound check for each dimension. Both *No Checks* and *Full Checks* are able to benefit from the VHPT walker. Finally, *Java ICRs* is our new method that competes with full compiler bounds checking.

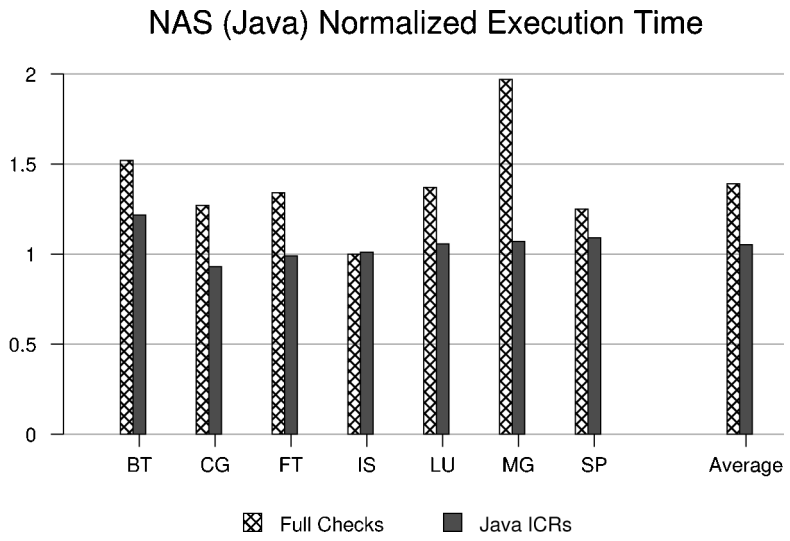

Fig. 7. Execution times for each program version on each of the NAS benchmarks (using linearized arrays). All times are normalized to the *No Checks* version, as it is the baseline; this means that smaller bars are better. Using *Java ICRs* is better than *Full Checks* in all programs. Note that the benchmarks are explained at the beginning of Section 5.

Note that *No Checks* is not legal according to the Java language specification—we use it only as a baseline to determine overheads.

Java ICRs is superior to *Full Checks* on all NAS Java benchmarks. The average normalized overhead of *Full Checks* is 39%, while the average overhead for *Java ICRs* is only 5%. The overhead of *Full Checks* primarily comes from extra instructions executed (compares) and extra cache reads (some of which are misses) to load the bounds. One anomalous result is that CG with *Java ICRs* is faster than CG with *No Checks*. Performance counters showed that *Java ICRs* had slightly fewer memory accesses; however, we could not determine why this occurred. Certainly, *No Checks* should be faster (and will be, in most instances).

As previously mentioned, the NAS benchmarks use linearized arrays. Each multidimensional array (in the original Fortran version) is transformed into a one-dimensional array (in the Java version). This was done intentionally by the NAS development team to reduce the cost of bounds checking [Frumkin et al. 2002]. Because Fortran references arrays in column-major order, linearizing arrays also avoids the need to interchange loops or transpose array dimensions for efficient execution in a row-major environment, such as Java. Both *Full Checks* and *Java ICRs* benefit from this conversion—the former because of fewer bounds checks (only one total check is needed) and the latter because of lower memory hierarchy overhead (only one total ICR per array is needed).

However, linearized arrays do not allow legitimate bounds checking. An access beyond the bound of the first dimension may not be detected, as it may fall in the allocated area of the array. We are currently working to delinearize the NAS Java Suite to fully test ICRs on them and currently have results from

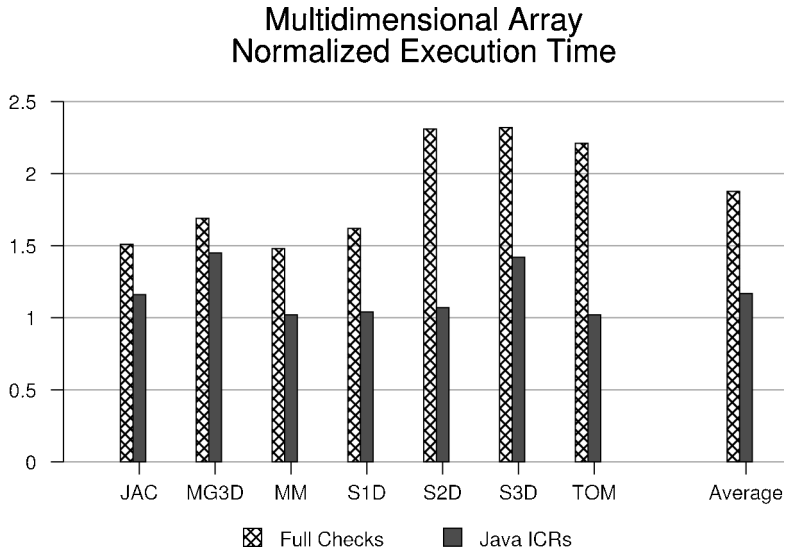

Fig. 8. Execution times for each program version on each of the hand-written benchmarks, as well as MG3D. All times are normalized to the *No Checks* version, as it is the baseline; this means that smaller bars are better. Using *Java ICRs* is better than *Full Checks* in all programs. Note that the benchmarks are explained at the beginning of Section 5.

Multigrid, which we denote MG3D. This program was produced by modifying `f2java` to (1) generate Java code using multidimensional arrays[4] and (2) transpose arrays to row-major order. Then, we modified, by hand, the main arrays to be four-dimensional, as in the NAS C version of MG, because the Fortran version of MG uses *common* blocks and `f2java` does not properly translate those. Because the translation is a time-consuming task—as a straight translation using `f2java` does not produce completely correct code—we also tested several representative multidimensional programs written by hand: MM, JAC, and TOM, as well as the three synthetic programs.

Figure 8 shows the execution times of our synthetic benchmarks, our hand-written scientific kernels, and MG3D. Notice that for the different synthetic versions the cost of *Full Checks* increases much faster than *Java ICRs* as dimensionality increases. This is because of the additional checking overhead caused by the increase in the number of dimensions. The increase in overhead of *Java ICRs* is because of stress in the memory hierarchy as a result of fragmentation, which causes an increase in the number of cache and TLB misses. The penalty for *Full Checks* on the three kernels averages 46%, while *Java ICRs* averages only 6%. MG3D is the worst performing benchmark with *Java ICRs* and it is still 24% better than *Full Checks* . This is strong evidence that the delinearized NAS suite will perform much better with *Java ICRs* than with *Full Checks* . Combined with the fact that the C versions of these programs (see Section 6) perform well with ICRs relative to explicit bounds checks, we believe that the NAS multidimensional Java programs will also perform well.

[4]The original `f2java` linearizes arrays.

It is important to note that the NAS multidimensional suite is almost a worst case for our ICR technique, because the NAS programs use blocking to improve locality. This decreases the size of each dimension, causing decreased memory system performance (see below).

One of the reasons for the significant overhead with *Full Checks* is that all checks (for all dimensions) occur in the innermost loop of a loop nest. Theoretically, it is possible, in some cases, to hoist invariant checks out of the innermost loop. Hence, we investigated the cause of this lack of code motion, keeping in mind that code motion can be difficult to implement because of aliasing. Because `gcj` is simply a front end to the `gcc` backend, we concluded that the backend of `gcc` usually cannot (or will not) hoist any checks in our benchmarks—a reminder that while algorithms for code motion are mature, in practice it can be hard to legally move code without risking modification of program semantics.

5.2 Low-Level Performance Details

The performance of *Java ICRs* is better than that of *Full Checks* in all of our benchmarks. However, a better understanding of the overheads caused by *Full Checks* and those caused by *Java ICRs* will help explain why, in general, *Java ICRs* performs so much better than *Full Checks* .

5.2.1 *Performance Counters.* We chose four of our test programs to examine in detail: TOM, S2D, S3D, and MG3D. For TOM and S2D, *Java ICRs* has almost no overhead, while *Full Checks* has over a factor of 2. For S3D and MG3D, *Java ICRs* has close to a 50% overhead for each, while *Full Checks* has about 120 and 60%, respectively. We examined the following counters for each program: total instructions executed, TLB misses, and all levels of cache (see Figure 12). This clearly shows that the reason for the poor performance on TOM for *Full Checks* is because of two reasons. First, there is an increase by about a factor of 2 in instructions as a result of bounds checks. The Itanium is a `VLIW` machine, creating bundles of instructions to make use of instruction-level parallelism (ILP). In many cases, the original code exhibited poor ILP, allowing bounds checking to be folded into the empty slots in the bundles. However, this was not always the case, so *Full Checks* still pays a heavy penalty for TOM. Second, the *Full Checks* versions have significantly more L1 accesses and misses. This is because of fetching the array length information for comparison.

While the time for *Java ICRs* is a vast improvement over *Full Checks* , ICRs do incur some overhead. Also shown in Figure 9 is the large increase in TLB misses for S3D and MG3D. In general, the degradation of the TLB performance when using ICRs is dependent on array size and array-access patterns. In particular, small array sizes increase fragmentation because each ICR must start on a new page. Typically, as the number of dimensions of an array increases, the size of each dimension tends to decrease. For example, Class W MG3D uses an array size of $64 \times 64 \times 64$, whereas TOM uses an array size of 768×768. The TLB will miss much more frequently compared to *Full Checks* (which packs consecutive rows) when array sizes are small. The cache misses do not increase significantly (other than for MG3D), mostly because of

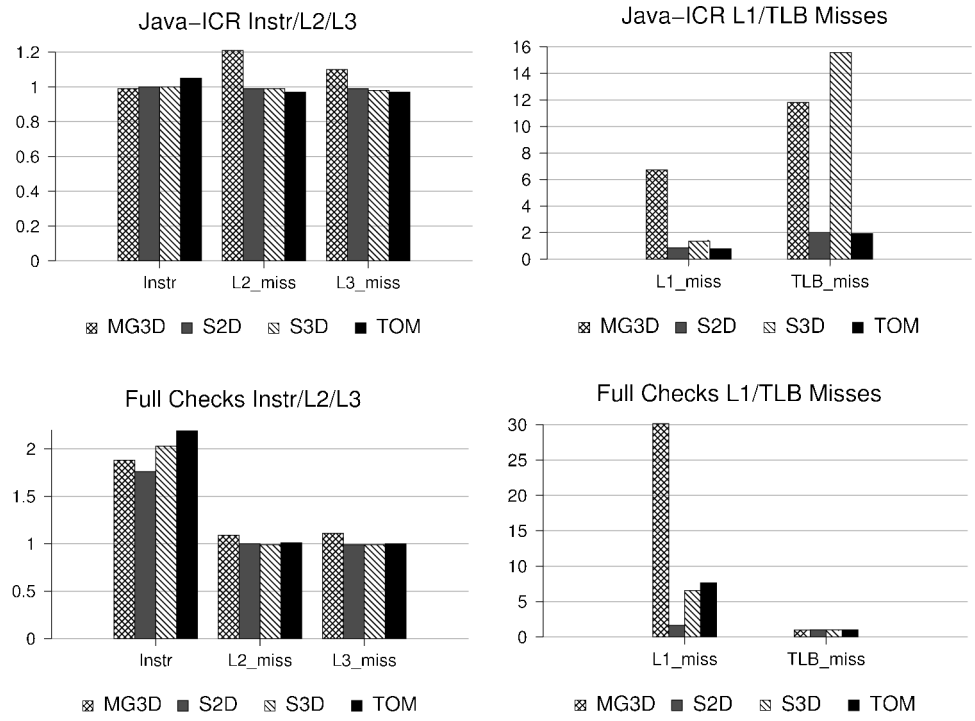

Fig. 9. Low-level performance counter results for *Full Checks* and *Java ICRs* for TOM, S2D, S3D, and MG3D. All times are normalized to the *No Checks* version.

the use of a 4 KB page—separate tests (not shown) revealed that using a 16 or 64 KB page size caused a large increase in L3 cache misses when using ICRs. The severe fragmentation from MG3D also causes performance degradation in all levels of the cache hierarchy. Both of these aspects are shown in Figure 9.

To further demonstrate the effect of the size of the last dimension on TLB performance, consider S2D and S3D. For S3D, which uses an array of size ($100 \times 100 \times 100$), *Java ICRs* has a 42% overhead compared to *No Checks*. Notice that the number of TLB misses for S3D *Java ICRs* (Figure 9) is 16 times more than *No Checks* and *Full Checks*. This is as a result of the small size (100) of the last dimension, which causes fragmentation. Because the size of each dimension tends to increase as the number of dimensions decreases, we chose an array size of 1000×1000 for S2D. With that size, S2D has an *Java ICRs* overhead of 6%. This large improvement over S3D is because of fewer TLB misses, as there is less internal fragmentation.

In general, we see a tradeoff between *Java ICRs* and *Full Checks*; the overhead of the former is in memory hierarchy overhead, while the overhead of the latter is in the increase in the number of instructions. Even with this substantial pressure on the memory hierarchy, *Java ICRs* significantly reduces the average penalty for performing bounds checks in Java.

5.2.2 *Effect of Memory Hierarchy Misses Versus Bounds Check.* Next, using S3D, we investigate the cost of misses in the memory hierarchy versus

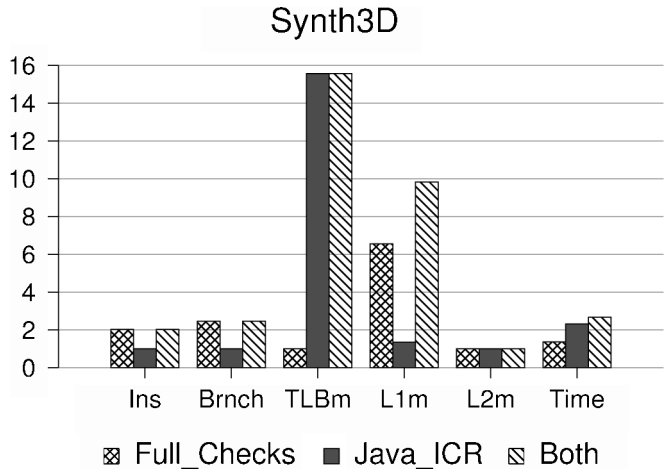

Fig. 10. Synth results. All times are normalized to the *No Checks* version.

bounds check. Specifically, we ran the usual tests using *Full Checks* and *Java ICRs* , but then added a test we call *Both* —this test uses both full bounds checking *and Java ICRs* . While one would always use *Full Checks* or *Java ICRs* , this test is useful to investigate where the overhead lies.

The results are shown in Figure 10. The overhead for *Both* is 166%, whereas the overhead for *Full Checks* and *Java ICRs* is 132 and 42%, respectively. For most of the key performance counters, it is clear that *Both* matches either *Full Checks* or *Java ICRs* . The one difference is in L1 cache misses, where *Both* has a greater number than *Full Checks* (and *Java ICRs* , but that is expected). This indicates the increased memory pressure caused by *Java ICRs* results in some additional L1 misses (in addition to the large number of TLB misses). However, this does not cause any difference in L2 misses. As Synth3D is nearly a worst-case application in terms of memory layout, we believe that *Java ICRs* results in additional TLB and (though not in our benchmarks) possibly L1 misses, while *Full Checks* results in additional instructions, branches, and L1 misses.

5.2.3 *Effect of Array Reference Reordering.* Finally, we investigate the effect of our GNU backend modifications. Recall that to avoid potentially unsafe Java code from being generated from our modified `gcj` , we had to limit the aggressiveness of the instruction scheduler. Specifically, we disallowed reordering of array references.

Table V shows results of different instruction scheduling schemes for the NAS benchmarks. The first column shows performance with a naive, yet correct, scheme to prevent reordering of array references. Those results are produced by completely disabling the instruction scheduler. As can be seen, this significantly degrades performance.

The second column shows full instruction scheduling, which results in good performance, but may result in a semantically illegal Java program. The third column gives performance of our Java implementation. As can be seen from the table, the degradation in performance is negligible (less than 4%) for most of

Table V. Effect of Different Instruction Scheduling Schemes[a]

Program	No Scheduling	Full Scheduling	Modified Scheduling
BT	90.5	55.0	66.9
CG	7.05	6.64	6.65
FT	5.63	4.05	4.06
IS	7.17	6.90	6.90
LU	280	172	175
MG	8.44	7.29	7.29
SP	263	177	184

[a]Times in seconds.

the programs. We note, however, that typical numerical workloads may have a higher cost, especially when aggressive optimization is done. For our programs, memory references were rarely, if at all, reordered by the instruction scheduler. However, our implementation ensures that such reordering does not occur.

Only one NAS program, BT, significantly benefits from array-reference reordering (which is only allowed in Java if it can be proved to be safe). Specifically, performance using our modified scheduler is about 20% worse than if full scheduling is used. This is likely because of the presence of large basic blocks containing primarily array references (e.g., the functions `add` and `x_solve`), which increases the probability that an unmodified instruction scheduler can (illegally) reorder a subset of these references.

6. APPLICATION TO OTHER LANGUAGES

This paper focuses on our ICR-based Java implementation, but ICRs are not specific to Java. In fact, ICRs are a general technique that can support any language that performs (or desires to perform) array-bounds checking. Two example languages are C# and Pascal. In addition, while not required in the language, for reliability it may be desirable to add bounds checking to C or Fortran; in fact, the latter has a compile-time option to do so. To demonstrate the generality of ICRs, we adapted them to support C. This section discusses the necessary changes to the implementation and the resulting performance.

Here we assume that if ICR technique is desired, then both ends of the array need to be checked (unlike Java, which actually only requires one check, as described earlier). Thus explicit bounds checks must make two, rather than one, check for each dimension.

6.1 Changes to ICRs

The same general idea with ICRs is used to support C, except that a library procedure is added to allocate space for arrays. All arrays are allocated as vectors of vectors, so that their memory layout are just as with Java multidimensional arrays.

Whereas in Java with ICRs an n-dimensional array access requires zero checks, the C version of ICRs requires one. We are able to reduce the $2n$ checks to one as follows. First, we align the end of an array to a page boundary, just as with Java ICRs; this immediately eliminates n bounds checks, one for each dimension. As an optimization, we observe that with a vector of vectors style

```
struct {
    void *value;           // address of data
    void *low_bound;       // low bound address
    void *high_bound;      // high bound address
} bounded_ptr_struct;
```

Fig. 11. Representation of pointers in bcc.

allocation, the values in the first $n-1$ dimensions are pointers. We avoid all bounds checks in these dimensions by zero filling all memory lying between the start of the mapped area and the start of the vector. Any dereference that occurs within this area will cause a NULL pointer exception. Of course, this cannot be applied to the last dimension, because it contains arbitrary objects. Still, this reduces the number of bounds checks to just one.

6.1.1 *Restrictions.* Unlike Java, the C version of ICRs has some restrictions. First, because the modern C standard allows array indices to be up to 2^{64}, there will exist programs (specifically, ones that make use of this feature) to which ICRs cannot be applied. In addition, the C-based ICRs will detect bounds violations immediately when assigning to a pointer any of the first $n-1$ dimensions of an array (e.g., $p = A[i]$, if p is a pointer and A is a two-dimensional array), because until that pointer is dereferenced, there will never be a protection violation.

We believe the first issue is not a significant problem for most C programs, as main memories on most systems are too small to make a 64-bit index practical. While our current implementation does not do so, to use ICRs we would need to statically prove that a 64-bit index was never used in a given array in order to apply ICRs (otherwise, we would use standard bounds checking on that array). For the second issue, we could explicitly issue a bounds check or generate a dereference. Our current implementation waits until the pointer is explicitly dereferenced to generate such an error. Such code is generally uncommon in scientific programs (the situation did not occur in our benchmarks).

6.2 Bounds Checking C Compiler

In our experiments, we generate all bounds checks with the bounds-checking compiler (bcc) [McGary], which is an extension to the GNU C compiler that performs full bounds checking. We ported bcc to the IA-64 architecture. For each pointer, bcc generates a three-member structure, which contains the memory location pointed to, as well as a lower and upper bound for the pointer (see Figure 11). These bounds are computed and set at allocation time by a special version of malloc. A dereference of the pointer results in generated code that ensures the dereferenced pointer is between the low and high bound set in the structure. When used in conjunction with ICRs (to provide a last-dimension check), a compiler flag is used to emit only the necessary check.

The bcc bounds checking code is efficient, as the code emitted is amenable to optimization. As mentioned earlier, the IA-64 architecture has three instruction bundles and, in many cases, there are empty (nop) slots in these bundles. Where possible, bcc will place bounds checking into such empty slots. Because

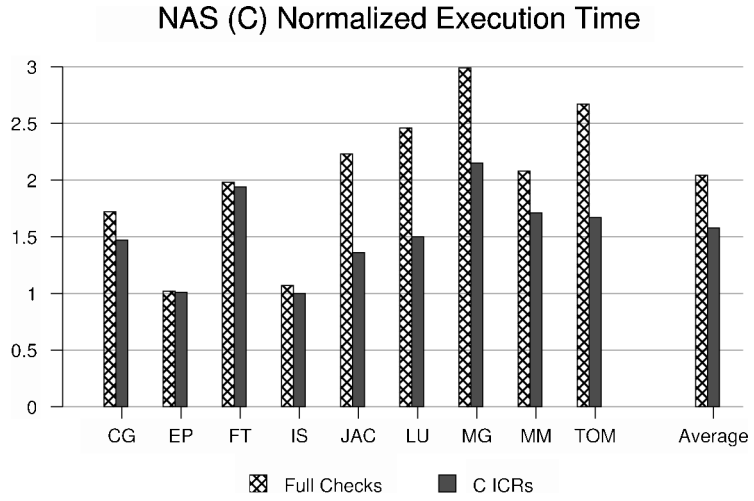

Fig. 12. Execution times for each program version on each benchmark. All times are normalized to the *No Checks* version, as it is the baseline; this means that smaller bars are better. Using *C ICRs* is better than *Full Checks* in all programs.

`bcc` changes pointers into structures, any library (e.g., `glibc`) that is linked with `bcc` code must also be compiled with `bcc` to ensure proper treatment of the pointer structures.

Our current implementation of `bcc` does not automatically handle statically allocated global arrays. We have modified the benchmarks by hand to transform such arrays into dynamically allocated arrays that are initialized at program startup. This transformation could be performed in the compiler, which would generate a list of arrays (and sizes) that must be allocated at startup time. Note that we do not change the `sizeof` operator, in that changing static arrays to dynamic ones will cause this operator to return the size of a pointer instead of the size of the array. This problem could be solved by applying the work in Dahn and Mancoridis [2003]. We note, however, that none of our benchmarks made use of the `sizeof` operator.

6.3 Performance of ICR-Based C

We used the same programs as with the Java implementation, with a few exceptions. We had to hand-write LU and MM, because `bcc` mishandles pointer arithmetic in the NAS versions and, hence, does not produce correct output.[5]

Figure 12 shows execution times of all of our benchmarks for each program version. All benchmarks run faster when using *C ICRs* than when using *Full Checks* . (Recall that when using *C ICRs* , one check is needed, as opposed to no checks with *Java ICRs* .) Overall, *C ICRs* averages 50% overhead, whereas *Full Checks* averages 96% overhead. It is important to note that the code generated by `bcc` for *Full Checks* is efficient, adding only a load, compare, and conditional

[5]The same bug occurred in BT and SP (the two other NAS benchmarks), but we did not hand-write them because they were much longer.

Table VI. Performance of Three Different Compilers on the NAS Suite Running on the Itanium 2, as well as Our Hand-Written Benchmarks[a]

Program	HotSpot	JRockit	gcj
BT	1508	360	82.9
CG	14.1	9.70	8.38
FT	33.5	14.6	5.49
IS	28.0	19.6	6.91
LU	7345	1790	229
MG	8.75	27.6	9.69
SP	4171	1560	222
TOM	990	121	53.4
MM	15.2	38.1	25.9
JAC	640	48.7	35.7

[a]Times in seconds. HotSpot uses the `-server` option, JRockit uses optimizations, and `gcj` uses -O3.

branch. Inspection of program counters showed that the overhead of *C ICRs* is, in general, fairly evenly divided between the cost of ICRs and the cost of the single bounds check. In the particular (worst) case of FT, *C ICRs* has an overhead of 88%, compared to 96% for *Full Checks* . For FT, which has many small arrays, most of this overhead is because of ICRs.

One of the reasons for the significant overhead with *Full Checks* is that all checks (for all dimensions) occur in the innermost loop of a loop nest. Theoretically, it is possible in some cases, to hoist invariant checks out of the innermost loop. However, while algorithms for code motion are mature, in practice it is hard to legally move code without risking modification of program semantics.

In summary, as expected, the overhead for both *Full Checks* and *C ICRs* increase when using C (compared to Java)—more checks are required in both. However, the relative difference in overheads is similar.

7. DISCUSSION

This section discusses several issues arising with this work. First, and most importantly, we discuss the legitimacy of using `gcj` , as opposed to a newer Java implementation such as HotSpot [SUN 2002] or JRockit [BEA 2005]. We believe that `gcj` compares at least equally with these compilers and especially favorably on the Itanium. Table VI shows results from executing several programs with all three compilers on the Itanium. Of note is that `gcj` has bounds checking enabled, whereas the other two can eliminate checks dynamically. Still, `gcj` greatly outperforms both compilers for all programs except MG and MM. Separate experiments show that `gcj` and HotSpot are competitive on Pentium-based architectures. It is beyond the scope of this paper to investigate why HotSpot and JRockit perform poorly on the Itanium, although we believe that it is possible that the backend on the Pentium has received more attention than its Itanium counterpart.

Second, we have discussed the advantage of using ICRs for reducing bounds-check overhead in Java. Again, ICRs are intended purely for scientific programs;

because the OS must use our *xvm* abstraction along with ICRs , there is inherent overhead (primarily because of additional TLB misses) in nonarray intensive parts of the code. The ramifications of this are that ICRs are appropriate for scientific codes, but not standard desktop Java applications, such as perhaps a database system. For example, when running the raytrace benchmark from the SPEC suite, which is not array intensive, ICRs added a 46% overhead. This is not surprising—it is currently up to the programmer to explicitly enable ICRs , and this should only be done for array intensive codes. In general, it would be possible to use static analysis to estimate when ICRs should be used. However, this is beyond the scope of this paper.

Next, we compare our new Java implementation, which uses ICRs within the `gcj` compiler to Ninja [Artigas et al. 2000]. Ninja works by creating *safe regions*. This not only improves performance by eliminating array-bound checks, but also allows more aggressive optimizations because an exception will not occur within a safe region. Our modified `gcj` , on the other hand, does not explicitly perform other optimizations, relying instead on standard `gcj` to perform those.

Several points are of note here. One is that our ICR -based technique is, in fact, orthogonal to the techniques used by Ninja . In particular, an unsafe region permits no aggressive optimization, including array-bound checks. In this case, our `gcj` will eliminate bounds checks. Thus, our ICRs could, in principle, be integrated into Ninja , providing a significant performance improvement for the large number of programs that defy static analysis. In fact, our manual inspection showed there were several such programs in the NAS suite (see Section 2). This is also confirmed by the reported results of the Ninja compiler itself, which was unable to cover a significant loop computation in TOM, resulting in poor performance [Artigas et al. 2000]. Furthermore, more complex programs are unlikely to receive complete coverage from safe regions.

Another issue is that our current implementation places all Java arrays in ICRs . In fact, only multidimensional arrays that are indexed in a row-wise manner are suitable for our technique (to avoid excessive TLB misses). We could extend our Java implementation to handle arrays accessed in a column-wise manner by either (1) transposing the array or (2) avoiding placing it in an ICR and performing traditional bounds checks. However, in the benchmarks used in this work, our current technique was sufficient.

Finally, as was discussed in Section 4, ICRs benefit from the use of a small page size. However, other applications may desire large page sizes. For example, some JVMs, such as HotSpot, map the heap using large pages [mic]. Fortunately, we do not believe that ICRs preclude large page sizes. In particular, new architectures provide multiple page sizes in hardware. An operating system can take advantage of this to allow different applications to use different page sizes, or even possibly different page sizes in the same application. There already exist Linux prototypes that allow the latter [Winwood et al. 2002]. Combined with our *xvm* , we argue that ICRs can use small page sizes with little, if any, effect on other applications.

8. CONCLUSION

This paper has introduced a new technique to check array-bounds *implicitly*, rather than the more traditional explicit way. We use compiler and operating system support to remove *all* bounds checks in Java programs and all but one check in C programs. The basic idea is to place each array object in a *index-confinement region* (ICR), which is an isolated virtual memory region. The rest of a ICR is unmapped and any access to that portion will cause a hardware protection fault.

In order to obtain this improvement, it was necessary to (1) create a Java implementation to perform special array allocations, as well as, (2) use a small (4-KB) page size with a large virtual address space. Combined, this reduces overhead in the cache as well as fragmentation in main memory because of program data, as well as page table data. We created a large virtual address space via an abstraction we called *xvm* , which provides an extended, customizable virtual memory, with little, if any, effect on other processes. In particular, in our Java benchmarks, ICRs average a small 9% overhead, while full compiler bounds checking averages nearly 63%. Overall, we believe that reducing the penalty for array-bounds checking in Java will make it, as well as other high-level languages, more attractive for parallel and HPC applications.

ACKNOWLEDGMENTS

We wish to thank David Mosberger for answering several questions about the Itanium Linux design and implementation.

REFERENCES

ARTIGAS, P., GUPTA, M., MIDKIFF, S., AND MOREIRA, J. 2000. Automatic loop transformations and parallelization for Java. In *Proc. ACM International Conference on Supercomputing*. 1–10.

BAILEY, D., BARTON, J., LASINSKI, T., AND SIMON, H. 1991. The NAS parallel benchmarks. RNR-91-002, NASA Ames Research Center.

BEA. 2005. BEA JRockit whitepaper (http://www.bea.com/content/news_events/white_papers/bea_jrockit_wp.pdf).

BODIK, R., GUPTA, R., AND SARKAR, V. 2000. ABCD: eliminating array-bounds checks on demand. In *Proc. ACM Conference on Programming Language Design and Implementation*. 321–333.

BOISVERT, R. F., DONGARRA, J. J., POZO, R., REMINGTON, K. A., AND STEWART, G. W. 1998. Optimizing array reference checking in Java programs. *Concurrency: Practice and Experience 10*, 11-13, 1117–1129.

CHASE, J. S., LEVY, H. M., FEELEY, M. J., AND LAZOWSKA, E. D. 1994. Sharing and protection in a single address space operating system. *ACM Transactions on Computer Systems 12*, 4 (May), 271–307.

CHIUEH, T. AND HSU, F. 2001. RAD: A compile-time solution to buffer overflow attacks. In *Proc. IEEE International Conference on Distributed Computing Systems*. 409–420.

COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., AND HINTON, H. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. USENIX Security Conference*. 63–78.

DAHN, C. AND MANCORIDIS, S. 2003. Using program transformation to secure c programs against buffer overflows. In *Working Conference on Reverse Engineering*. 323–333.

FRUMKIN, M., SCHULTZ, M., JIN, H., AND YAN, J. 2002. Implementation of the NAS parallel benchmarks in Java. NAS-02-009, NASA Ames Research Center.

GUPTA, R. 1993. Optimizing array-bound checks using flow analysis. *ACM Letters on Programming Languages and Systems 2*, 1–4 (Mar.–Dec.), 135–150.

INTEL. 2006. Intel Itanium Architecture Software Developer's Manual (http://download.intel.com/design/itanium/manuals/24531705.pdf).

ITZKOVITZ, A. AND SCHUSTER, A. 1999. Multiview and Millipage—fine-grain sharing in page-based DSMs. In *Proc. USENIX Operating Systems Design and Implementation*. 215–228.

Java hotspot vm options (http://java.sun.com/docs/hotspot/vmoptions.html).

KOLTE, P. AND WOLFE, M. 1995. Elimination of redundant array subscript range checks. In *Proc. ACM Conference on Programming Language Design and Implementation*. 270–278.

LAM, L. AND CHIUEH, T. 2005. Checking array-bound violation using segmentation hardware. In *Proc. IEEE International Conference on Dependable Systems and Networks*. 388–397.

MARKSTEIN, V., COCKE, J., AND MARKSTEIN, P. 1982. Optimization of range checking. In *Proc. ACM Conference on Programming Language Design and Implementation*. 114–119.

MCGARY, G. Bounds-checking C compiler (http://www.gnu.org/software/gcc/projects/bp/main.html).

MIDKIFF, S. P., MOREIRA, J. E., AND SNIR, M. 1998. Optimizing array reference checking in Java programs. *IBM Systems Journal 37*, 3, 409–453.

MOREIRA, J., MIDKIFF, S. P., GUPTA, M., ARTIGAS, P., AND SNIR, M. 2000. Java programming for high performance numerical computing. *IBM Systems Journal 39*, 1, 21–56.

MOSBERGER, D. AND ERANIAN, S. 2002. *IA-64 Linux Kernel: Design and Implementation*. Prentice-Hall, Eaglewood Cliffs, NJ.

PERENS, B. Electric Fence (http://sunsite.unc.edu/pub/linux/devel/lang/c/electricfence-2.0.5.tar.gz).

RUGINA, R. AND RINARD, M. C. 2000. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proc. ACM Conference on Programming Language Design and Implementation*. 182–195.

SUN. 2002. The Java HotSpot virtual machine (http://java.sun.com/products/hotspot/docs/whitepaper/java_hotspot_v1.4.1/jhs_141_wp_d2a.pdf).

TALLURI, M., HILL, M. D., AND KHALIDI., Y. A. 1995. A new page table for 64-bit address spaces. In *Proc. ACM Symposium on Operating Systems Principles*. 184–200.

WINWOOD, S., SHUF, Y., AND FRANKE, H. 2002. Multiple page size support in the linux kernel.

XI, H. AND PFENNING, F. 1998. Eliminating array-bound checking through dependent types. In *Proc. ACM Conference on Programming Language Design and Implementation*. 249–257.

XI, H. AND XIA, S. 1999. Towards array-bound check elimination in Java virtual machine langauge. In *Proc. Centre for Advanced Studies Conference*. 110–125.

YOUNG, M., AVADIS TEVANIAN, J., RASHID, R., EPPINGER, D. G. J., CHEW, J., BOLOSKY, W., BLACK, D., AND BARON, R. 1987. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proc. ACM Symposium on Operating Systems Principles*. 63–76.

Received May 2005; revised February 2006 and May 2006; accepted May 2006