Implicit Java Array Bounds Checking on 64-bit Architectures*

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

ABSTRACT

Interest in using Java for high-performance parallel computing has increased in recent years. One obstacle that has inhibited Java from widespread acceptance in the scientific community is the language requirement that all array accesses must be checked to ensure they are within bounds. 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 new Java implementation 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 our new Java implementation reduces the overhead of bounds checking from an average of 63% to an average of 9% on our benchmarks.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*optimization*

General Terms

Measurement, Performance

Keywords

Array-Bounds Checking, Java, Virtual Memory

1. INTRODUCTION

Interest is growing in the scientific programming community in using Java for high performance computing (HPC). Some reasons for this include attractive features such as portability, expressiveness, and safety. In addition, threads are built into the Java language. Finally, rapid progress is being made in Java compiler technology.

However, several obstacles currently prevent Java from becoming widely accepted for HPC. One such obstacle is the Java language specification requirement that all array accesses be checked to ensure they are within bounds. This often causes a run-time performance penalty, because in general, a Java compiler must ensure all array references are within bounds by adding explicit checks preceding an array access. 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.

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 Java to have any chance to be widely used for HPC applications, array bounds checking overhead must be alleviated.

The current state of the art for reducing bounds checking overhead is to use static analysis to eliminate explicit checks by proving an array reference is within its bounds [4]. 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 [2]. Second, even when static analysis is possible, currently available compilers may have difficulty removing explicit checks.

Rather than attempting to eliminate explicit bounds checks statically, our goal is to use compiler and operating system support to implicitly check array bounds in Java programs. 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 increasingly are difficult to analyze statically. If static analysis cannot eliminate bounds checks, then the compiler must insert n bounds checks for an n-dimensional array reference. Instead, we perform no static analysis of the program, and we insert zero checks for array bounds. We place each array object in an *index confinement region* (ICR). An ICR 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 32GB ICRs, each of which is large enough to implicitly catch any illegal access to an array of double-precision numbers.

We made two primary modifications to `gcj`, the GNU Java im-

*This research was supported in part by a State of Georgia *Yamacraw* grant as well as NSF Grant No. 0234285.

ICS'04, June 26–July 1, 2004, Saint-Malo, France.

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

Table 1: Percentage of bounds checks, for four NAS programs, that we believe could be eliminated statically. In practice, we are not aware of a Java compiler that can successfully eliminate *all* of the removable checks indicated above.

plementation. 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.

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 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 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 provides implementation details, and Section 4 presents performance results. Section 5 discusses issues arising in this work. Finally, Section 6 concludes.

2. RELATED WORK

A significant body of work exists on static analysis to eliminate array bounds checks. Kolte and Wolfe [12] perform partial redundancy analysis to hoist array bound checks outside of loops. Their algorithm was based on that described by Gupta [10], who formulated the problem as a dataflow analysis. In ABCD [4], Bodik et al. implemented a demand-driven approach to bounds checking in Java. Artigas et al. [1] addresses the problem by introducing a new array class with the concept of a *safe region*. Xi and Pfenning [17] introduce the notion of dependent types to remove array bound checks in ML; Xi and Xia [18] extend this idea to Java. Rugina and Rinard [15] provide a new framework using inequality constraints for dealing with pointers and array indices, which works for both statically and dynamically allocated areas.

All the above analyses are performed at compile time. This has the advantage of avoiding run-time overhead but fails when either (1) the code 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 1 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 Java compiler that will successfully eliminates *all* removable checks. In these cases compile-time schemes must fall back to general run-time checking. Instead, our Java 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 [6]. 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 [14], 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 [11], 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; then, both pages are mapped to the same physical page. Different protections can then be used on each variable, because protections are done at the virtual page level.

Our extended and customizable virtual memory abstraction (*xvm*) [3] 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 [19], 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, [5] describes Opal, which places all processes in a single 64-bit virtual address space. This allows for a more flexible protection structure.

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 [8]. Another compile-time solution, RAD [7], involves modifying the compiler to store return addresses in a safe location. This solution retains binary compatibility because stack frames are not modified.

3. 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.

3.1 Index Confinement Regions

An Index Confinement Region (ICR) is a large, isolated region of virtual memory. When an array is placed in an appropriately-

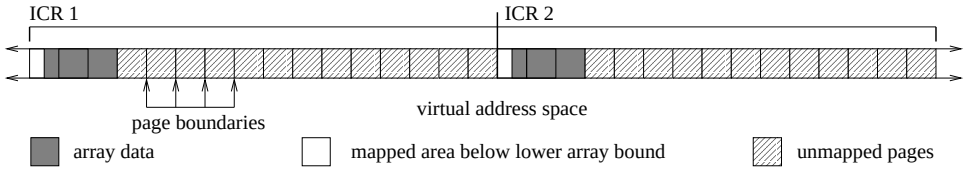

Figure 1: Two index confinement regions. Each array is isolated from any other program data; bounds checking is done automatically through unmapped pages.

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 16GB 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 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 4GB (2^{32}) and the size of the array element type; this can be calculated at allocation time. Although each ICR is several GB in size, a 64-bit virtual address space permits the allocation of millions of ICRs.

Figure 1 shows two regions and their associated arrays. In an ICR, pages in which the actual array resides are mapped with read and write permission. 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[1]. 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.

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 runtime system must be modified to place arrays in ICRs. Additionally, 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 runtime libraries to facilitate the ICR-based allocation of Java arrays.

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

3.2 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.

3.2.1 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 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 due to 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 2). While this is not a problem in standard `gcj` because an explicit comparison is made to the upper bound, our implementation does 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 runtime 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`.

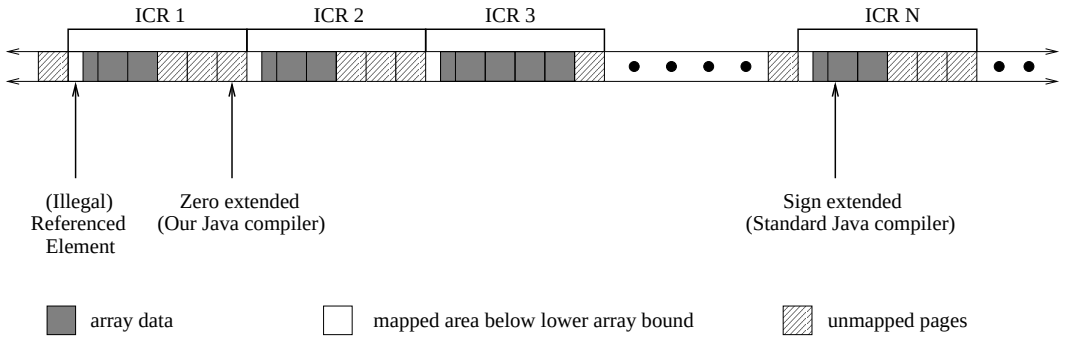

Figure 2: Pictorial description of an illegal array reference and its corresponding sign-extended index expression versus zero-extended index expression with ICRs.

Because all ICR support is implemented in the compiler, runtime 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 3.3 discusses the impact of ICRs on the memory hierarchy along with OS support for ICRs.

3.2.2 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.

3.3 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, though the address space available in Linux (and hence the number of ICRs that can be used) decreases with the page size due to 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* [3] 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

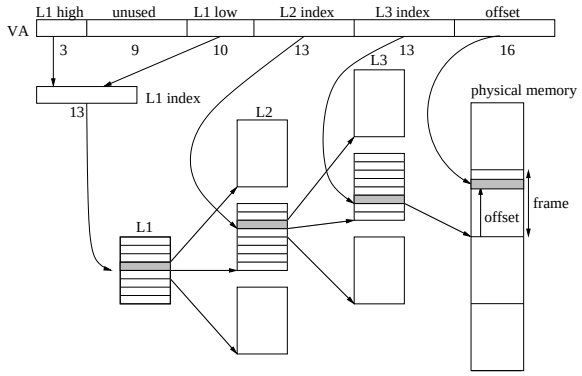

Figure 3: Address space layout and address translation for regular processes with a three-level page table in Linux, using 4KB pages.

directory, so 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 3, largely borrowed from [13]), a 4KB page size provides an 320GB address space, which is far too small for any of our benchmarks, while a 64KB page size provides 20PB of address space, which is sufficient for all of our benchmarks. We modified the 4KB kernel directory structure to allow the large virtual address space allowed by the 64KB kernel, while maintaining for ICRs the cache and memory benefits of the 4KB 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 32PB, which is adequate space for hundreds of thousands of ICRs. Figure 4 shows the new scheme.

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 4KB page size directory contains 512 entries, so in an extended virtual address space as described above, each L3PD contains 256K entries (512 pages $\times$ 512 entries per page), assumed to represent *contiguous* virtual memory. Hence, one L3PD represents 1GB (256KB $\times$ 4KB) of virtual memory. Unfortunately, arrays placed in ICRs are at least 4GB from one another. This means that when using the scheme

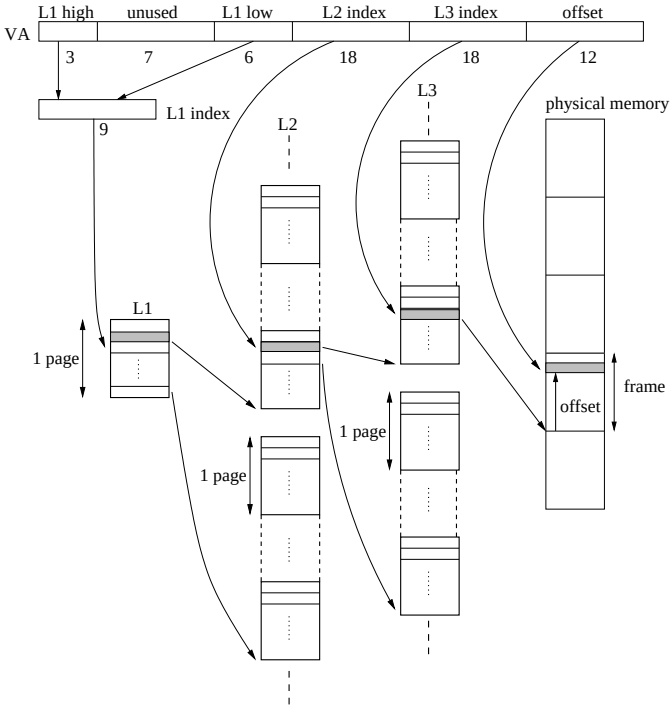

Figure 4: Address space layout for processes using our *xvm* abstraction. The L2PD and L3PD are each 512 pages instead of 1 page.

shown in Figure 4, 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 due to directories is increased considerably. This is a well-known problem with extremely sparse address spaces, which are generally better served using a hashed or clustered page table [16].

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. Without our customized addressing scheme, the kernel runs out of memory due to the large number of directory allocations in Multigrid and Fourier Transform. With our scheme, both programs run successfully.

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 runtime library), which converts a standard Linux 3-level page table into an multipage, multilevel extended version (as shown in Figure 4). This allows only those processes that make use of ICRs to incur the memory overhead due to the larger directories when using *xvm*. Further details on our *xvm* design and implementation can be found in [3].

4. 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 [9], 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 seconds. In addition, we include 3 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 900MHz Itanium-2 with 1.5GB memory, a 16KB L1 instruction cache, 16KB L1 data cache, 256KB L2 cache, and 1.5MB L3 cache. The operating system is Debian Linux version 2.4.19. We use wallclock 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. Then, we further examine some of the results through inspection of hardware-level counters.

4.1 Overall Execution Times

Figure 5 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. Finally, *Java ICRs* is our new method that competes with full compiler bounds checking. 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 overheard 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.

As previously mentioned, the NAS benchmarks use linearized arrays. Each multidimensional array (in the original Fortran version) is transformed into a 1-dimensional array (in the Java version). This was done intentionally by the NAS development team to reduce the cost of bounds checking [9]. Also, because Fortran references arrays in column-major order, linearizing arrays 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 de-linearize the NAS Java Suite to fully test ICRs on them and currently have results from Multigrid, which we denote MG3D. This program was produced by modifying `f2java` to (1) generate Java code using multidimensional arrays[2] 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, TOM, as well as the three synthetic programs.

Figure 6 shows the execution times of our synthetic benchmarks, our hand-written scientific kernels, and MG3D. Notice that for the different synthetic versions that the cost of *Full Checks* increases much faster than *Java ICRs* as dimensionality increases. This is due to the additional checking overhead caused by the increase in the number of dimensions. The increase in overhead of *Java ICRs* is due to stress in the memory hierarchy due to 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 if *Java ICRs*, and it is still 24% better than *Full Checks*. This is strong evidence that the de-linearized NAS suite will perform much better with *Java ICRs* than with *Full Checks*. Combined with the fact that our previous study showed that the C versions of these programs [3] perform well with ICRs relative to explicit bounds checks, we believe that the NAS multidimensional Java programs will also perform well.

It is important to note that the NAS multidimensional suite is almost a worst case for our ICR technique, because the NAS programs use tiling 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 due to aliasing. Because `gcj` is simply a front end to the `gcc` backend, we concluded that the backend of `gcc` usually cannot 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.

4.2 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*.

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 two. 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 7). 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 due to 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 due to 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 7 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 due to the use of a 4KB page—separate tests (not shown) revealed that using a 16KB or

[2] The original `f2java` linearizes arrays.

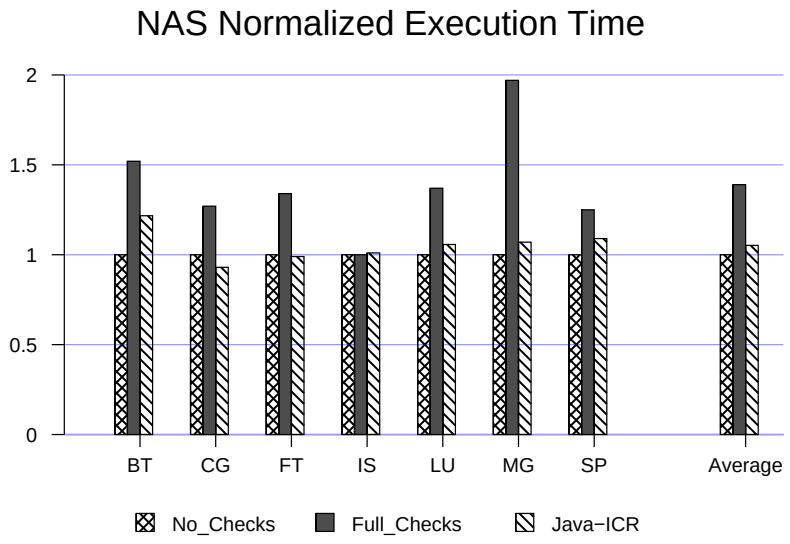

Figure 5: Execution times for each program version on each of the NAS benchmarks (using linearized arrays). All times are normalized to the *No Checks* version, as it is the baseline; this means that smaller bars are better. Using *Java ICRs* is better than *Full Checks* in all programs. Note that the benchmarks are explained at the beginning of Section 4.

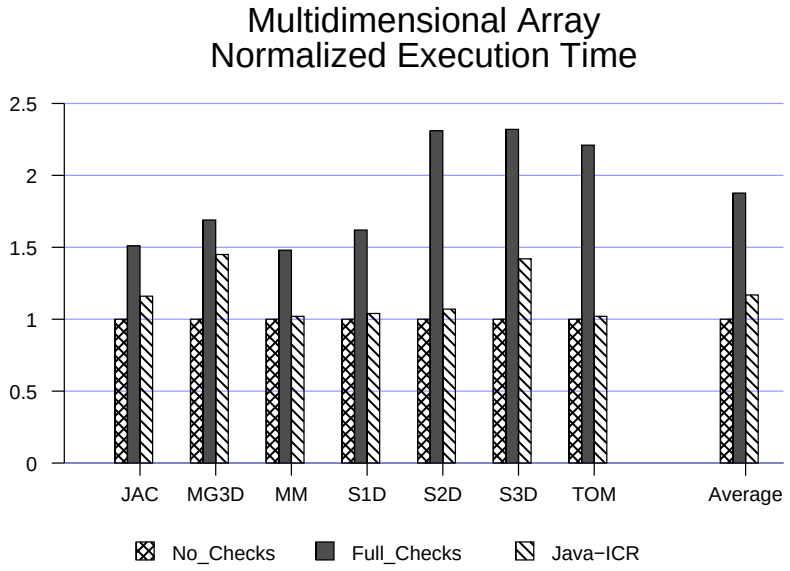

Figure 6: Execution times for each program version on each of the hand-written benchmarks as well as MG3D. All times are normalized to the *No Checks* version, as it is the baseline; this means that smaller bars are better. Using *Java ICRs* is better than *Full Checks* in all programs. Note that the benchmarks are explained at the beginning of Section 4.

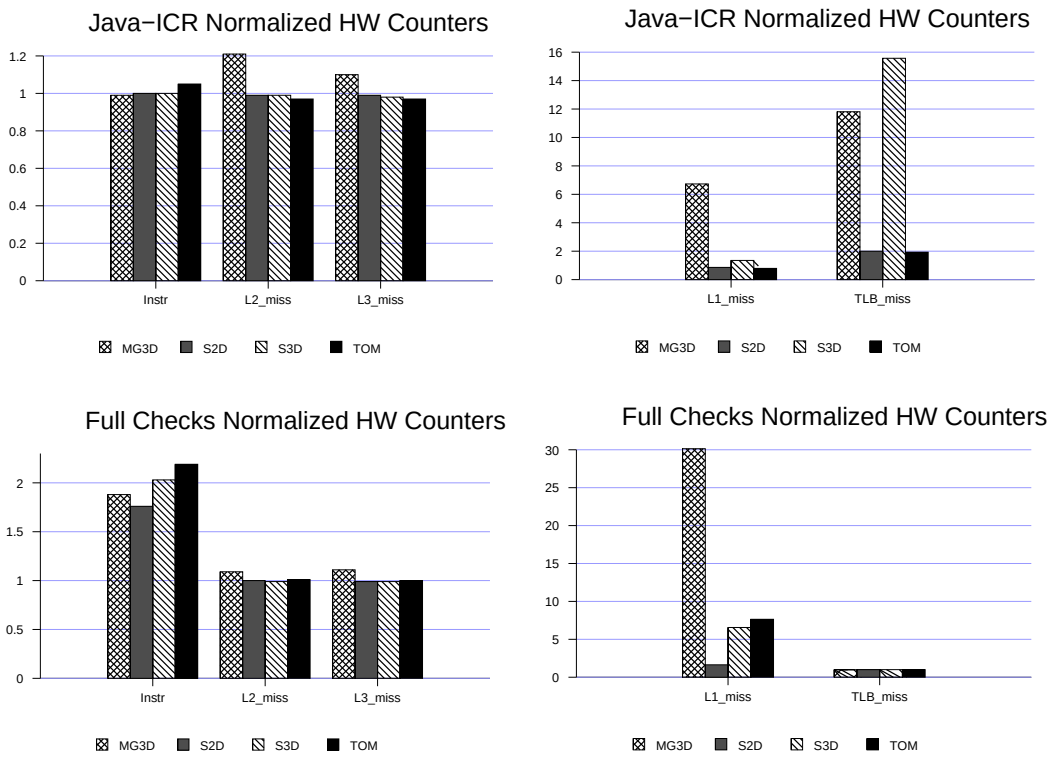

Figure 7: Low-level performance counter results for *Full Checks* and *Java ICRs* for TOM, S2D, S3D, and MG3D. All times are normalized to the *No Checks* version.

64KB 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 7.

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 7) is 16 times more than *No Checks* and *Full Checks*. This is due to 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 due to 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.

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 2 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 degrades performance significantly.

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

Table 2: Effect of different instruction scheduling schemes (times in seconds).

The second column shows full instruction scheduling, which results in good performance but may result in an incorrect 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 the programs. This indicates that memory references were rarely, if at all, reordered by the instruction scheduler. However, our implementation ensures that such reordering does not occur.

Only one program, BT, benefits significantly from array reference reordering (which is illegal in Java). Specifically, performance using our modified scheduler is about 20% worse than if full scheduling is used. This is likely due to 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.

5. DISCUSSION

This section discusses two main issues arising with this work. First, we compare our new Java implementation, which uses ICRs within the `gcj` compiler, to Ninja [1], the state of the art in bounds checking for Java. Ninja works by finding *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 `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. So, 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[1]. Furthermore, more complex programs are unlikely to receive complete coverage from safe regions.

Second, our current implementation places all Java arrays in ICRs. In fact, only multi-dimensional 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.

6. CONCLUSION

This paper has introduced a new technique to check array bounds in Java programs *implicitly*, rather than the more traditional explicit way. We use compiler and operating system support to remove *all* bounds checks in Java programs. This means that instead of n bounds checks for an n-dimensional array reference, zero checks are inserted. 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. Combined with the array size restriction in Java, we are able to conform to Java semantics without adding any bounds checks.

In order to obtain this improvement, it was necessary to (1) create a new Java implementation to perform special array allocations as well as (2) use a small (4KB) page size with a large virtual address space. Combined, this reduces overhead in the cache as well as fragmentation in main memory due to 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, 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 Java a more attractive language for parallel and HPC applications.

Acknowledgements

We wish to thank David Mosberger for answering several questions about the Itanium Linux design and implementation. In addition, we would like to thank the anonymous referees for their helpful comments.

7. REFERENCES

[1] P. Artigas, M. Gupta, S. Midkiff, and J. Moreira. Automatic loop transformations and parallelization for Java. In *International Conference on Supercomputing*, pages 1–10, May 2000.

[2] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. RNR-91-002, NASA Ames Research Center, Aug. 1991.

[3] C. Bentley, S. A. Watterson, and D. K. Lowenthal. Operating system support for low-cost array bounds checking on 64-bit architectures. Technical report, University of Georgia, Nov. 2003.

[4] R. Bodik, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–333, 2000.

[5] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single address space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, May 1994.

[6] T. Chiueh. Personal communication. Oct. 2002.

[7] T. Chiueh and F. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *International Conference on Distributed Computing Systems*, Apr 2001.

[8] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, Jan 1998.

[9] M. Frumkin, M. Schultz, H. Jin, and J. Yan. Implementation of the NAS parallel benchmarks in Java. NAS-02-009, NASA Ames Research Center.

[10] R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1-4):135–150, March–December 1993.

[11] A. Itzkovitz and A. Schuster. Multiview and Millipage - fine-grain sharing in page-based DSMs. In *Operating Systems Design and Implementation*, pages 215–228, 1999.

[12] P. Kolte and M. Wolfe. Elimination of redundant array subscript range checks. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 270–278, 1995.

[13] D. Mosberger and S. Eranian. *IA-64 Linux Kernel: Design and Implementation*. Prentice-Hall, 2002.

[14] B. Perens. Electric Fence (http://sunsite.unc.edu/pub/linux/devel/lang/c/electricfence-2.0.5.tar.gz).

[15] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 182–195, June 2000.

[16] M. Talluri, M. D. Hill, and Y. A. Khalidi. A new page table for 64-bit address spaces. In *Symposium on Operating Systems Principles*, Dec. 1995.

[17] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, 1998.

[18] H. Xi and S. Xia. Towards array bound check elimination in Java virtual machine langauge. In *CASCON '99*, pages 110–125, 1999.

[19] M. Young, J. Avadis Tevanian, R. Rashid, D. G. J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *11th Symposium on Operating Systems Principles*, Nov. 1987.