LOW COST ARRAY BOUNDS CHECKING FOR 64-BIT ARCHITECTURES

by

CHRISTOPHER W. BENTLEY

(Under the direction of David K. Lowenthal)

ABSTRACT

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 performance limitation is a significant obstacle preventing the scientific community from adopting compiler-enforced array bounds checks.

To reduce the overhead, we have created an abstraction that called Index Confinement Regions (ICRs). The basic idea is to place an array into a very large virtual memory region, such that any reference to the array is confined to the region. Only the portion of the ICR corresponding to the array is permissible to access. ICRs reduce the number of necessary bounds checks for n-dimensional array access from $2n$ to 1 for C, and from n to 0 for Java, yielding a significant reduction in execution time for array-intensive applications.

INDEX WORDS: Array, Bounds Check, Index Confinement Region, 64-bit architectures, Linux

LOW COST ARRAY BOUNDS CHECKING FOR 64-BIT ARCHITECTURES

by

CHRISTOPHER W. BENTLEY

B.S., The University of Georgia, 2001

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2003

Low Cost Array Bounds Checking for 64-bit Architectures

by

Christopher W. Bentley

Approved:

Major Professor: David K. Lowenthal

Committee: Scott A. Watterson
Suchendra M. Bhandarkar

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2003

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 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 require checking of array bounds. Despite this 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, such as Java, require array bounds checking.

While array accesses are infrequent in some applications, our focus is on scientific programs. Such applications tend to be array intensive, which means that checking array bounds can significantly increase execution time. It is this overhead that keeps many programmers from using languages such as Java for scientific applications. If Java is to be accepted for such a purpose, the overhead associated with bounds checking must be alleviated.

While in some programs static analysis can eliminate checks by proving an array reference is within its bounds, the dynamic nature of many scientific codes makes this difficult or impossible. Furthermore, even when static analysis is possible, currently available compilers have difficulty removing checks. For example, empirical testing of ABCD [1], a state-of-the-art Java bounds check elimination scheme, showed that only a small portion of the bounds checks in the NAS Java Benchmark Suite [2] were removed.

Rather than attempting to eliminate bounds checks statically, our goal is to use operating system support to significantly reduce the cost of full array bounds checking. Furthermore, we

wish to do this using currently available technology (existing hardware and operating systems), as opposed to proposing new hardware or designing a completely new operating system.

In this thesis, we reduce array bounds checking cost through a new abstraction that we call index confinement regions (ICRs). An ICR is an isolated virtual memory region. Only a small portion of the ICR 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. 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. References to elements of arrays of integers or doubles can be confined within such ICRs as long as they use no more than 32-bit index expressions, as many languages require.

The ICR abstraction is applicable to C and Java arrays and can greatly reduce the overhead for performing bounds checking. For example, once a C array is placed within an ICR, only one bound in the last dimension must be checked. This means that for an n-dimensional array reference, instead of inserting $2n$ checks (as would be strictly required), we need to insert only one check. Also, the Java language specification allows certain optimizations that permit Java arrays to be fully protected in an ICR with zero explicit bound checks rather than the n bounds checks that are normally inserted. This has potential for large speedup for programs in which array references consume a significant portion of the overall execution time. The ICR abstraction is completely general; multidimensional arrays can be protected by representing them as a vector of vectors, where each vector is placed in an ICR. In fact, Java arrays are already represented in this manner since there is no true multi-dimensional array in Java.

Because processes that utilize ICRs make sparse use of their address space, the reduction in bounds checking cost can be potentially overshadowed by memory hierarchy overhead. This overhead includes the TLB (more pages are used), the cache (conflicts are possible), and main memory (internal fragmentation exists). Furthermore, the amount of physical memory occupied by page tables increases.

Accordingly, we modified the Linux kernel to allow processes to create an extended, customizable virtual memory that we call *xvm*. An *xvm* is extended because it allows a large address space

with a small page size through a multipage, multilevel directory scheme. Also, an *xvm* is customizable because it allows a process to choose an alternative virtual addressing scheme that is more amenable to its access patterns. In the specific case of ICRs, both of these attributes are important. First, a large address space (several petabytes) is required for ICRs, but a small page size reduces internal fragmentation and makes better use of the cache. Second, while ICRs have sparse access patterns, they are regular and hence can be optimized to decrease the memory consumed by page tables. It is important to note that the use of an *xvm* by one process does *not* affect any other process.

Our results on a 900 MHz Itanium-2 show that performance of programs that use ICRs is superior to those obtained with programs that use full bounds checking with a state-of-the-art bounds checking compiler, such as `gcj` or `bcc` [3]. In particular, our approach incurs 46% less overhead on average than full bounds checking for the scientific benchmarks we tested that are written in C and 54% less overhead for programs written in Java. In addition, *all* of the benchmarks performed better using ICRs with 4KB pages than with full bounds checking.

The remainder of this thesis is organized as follows. The next section describes related work. Section 3 provides implementation details, and Section 4 presents performance results. Then, Section 5 discusses issues that arise in this work and presents possible avenues for future work. Finally, Section 6 concludes.

CHAPTER 2

RELATED WORK

A significant body of work exists on static analysis to eliminate array bounds checks. Kolte and Wolfe [4] perform partial redundancy analysis to hoist array bound checks outside of loops. Their algorithm was based on that described by Gupta [5], who formulated the problem as a dataflow analysis. In ABCD [1], Bodik et al. implemented a demand-driven approach to bounds checking in Java. Artigas et al. [6] addresses the problem by introducing a new array class with the concept of a *safe region*. Xi and Pfenning [7] introduce the notion of dependent types to remove array bound checks in ML; Xi and Xia [8] extend this idea to Java. Rugina and Rinard [9] 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 analysis 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, our inspection of the CG, FT, and MG benchmarks from the NAS suite show that it is extremely difficult and impractical to prove that array references are within bounds. Furthermore, our empirical examination of the code produced by ABCD for each program in the NAS suite shows that in practice, many checks are not removed statically. In these cases compile-time schemes must fall back to general run-time checking. Instead, ICRs decrease the cost of bounds checking and do not depend on static analysis to do so.

A similar approach to ours is to use segmentation for no-cost bounds checking [10]. 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 [11], 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, ICRs are able to catch any illegal reference.

Our approach bears some similarities to Millipede [12], 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*) [13] 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 [14], 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, [15] describes Opal, which places all processes in a single 64-bit virtual address space. This allows for a more flexible protection structure.

CHAPTER 3

IMPLEMENTATION

This section describes implementation details on an Itanium 2, which is a 64-bit machine. First, the implementation of index confinement regions (ICRs) for C is discussed along with modifications to the Bounds Checking C Compiler, `bcc`. Then, ICRs for Java and modifications to the GNU Java Compiler, `gcj` are described. Finally, ICRs pose several challenges for the IA-64 Linux kernel. Accordingly, modifications to the kernel to face these challenges are detailed.

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-sized ICR, references to this array are confined within the ICR. For example, consider placing a one-dimensional integer array, indexed by signed 32-bit expressions, within an ICR. If the size of the ICR is chosen to be at least 16GB and the array is placed in the center of the ICR, it is impossible to generate an array reference that is outside of the ICR. If the compiler treats the index as signed, any attempt to reference an element beyond the upper bound of the ICR will result in arithmetic overflow and produce a reference in the lower half of the ICR, which is intended for negative indices. Thus, an ICR must be large enough so that any 32-bit index expression will result in an access within the 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.

Specific usage and support for ICRs differ depending on the target language. Below, the application of ICRs is described for C. Then, `bcc`, the Bounds Checking C Compiler is introduced along with modifications necessary to facilitate bounds checking with ICRs. Afterward, the implementation of ICRs is described for Java.

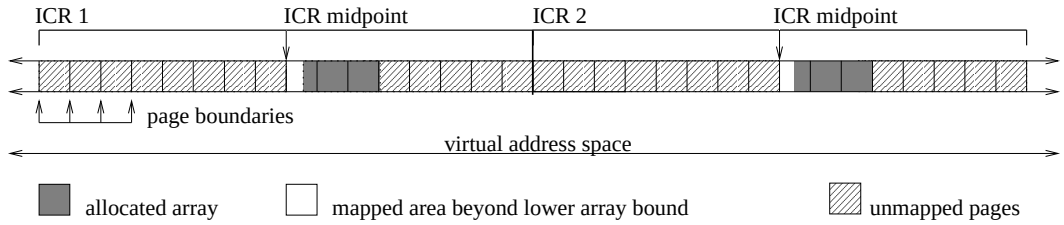

Figure 3.1: Two index confinement regions for C arrays. Each array is placed in the center of the ICR and is isolated from any other program data; bounds checking is done automatically through unmapped pages.

3.2 ICRs FOR C

Figure 3.1 shows two ICRs and their associated C 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 by calling `mmap` with (1) the target address of the next available ICR and the (2) the `MAP_FIXED` flag. This allows partial automatic bounds checking, because an access to any unmapped page results in a protection violation. Because in general the array size is not a multiple of the page size, one end will not be fully protected; hence, one explicit bounds check is necessary. We align arrays so that the upper limit of the array is at a page boundary and explicitly check the lower bound[1]. This is also shown in Figure 3.1. Also, regions are pooled to eliminate calls to `mmap` whenever possible.

The ICR abstraction extends naturally to n dimensions. This is done by allocating n-dimensional arrays as vectors of vectors, placing each vector in a distinct ICR. This immediately eliminates n bounds checks, one for each dimension. As an optimization, we observe that with a vector of vectors style 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. This cannot be applied to the last dimension, because it contains arbitrary

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

objects. Still, our optimization reduces the number of bounds checks from what would be $2n$ with a bounds checking compiler to 1 with ICRs.

This does impose one restriction on array accesses in this framework: bounds violations will not be detected 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. This could be handled in the compiler, either by explicitly issuing a bounds check or by generating 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)

To facilitate ICRs for C, `bcc` must be modified to emit a single bounds check for the last dimension of any array access. In the following section, the method `bcc` uses to check array bounds is introduced along with changes for ICRs.

3.3 Bounds Checking C Compiler

In all C experiments, bounds checking code is compiled with the Bounds Checking Compiler (`bcc`), which is an extension to the GNU C Compiler that performs full bounds checking. First, `bcc` was ported to the IA-64 architecture. The `bcc` is able to perform bounds checking by storing additional information for pointers. For each pointer, `bcc` generates a 3 member structure, which contains the memory location pointed to, as well as a lower and upper bound for the pointer. 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. On the IA-64, a single bounds check incurs a load, a compare, and a branch instruction. When used in conjunction with ICRs (to provide a last-dimension low bound check), a compiler flag is used to emit only the necessary check, rather than 2 checks for each dimension.

The `bcc` bounds checking code is efficient, as the code emitted is amenable to optimization. The Itanium is a `VLIW` machine, creating bundles of instructions to make use of instruction-level parallelism (ILP). In many cases there are empty (`nop`) slots in these bundles. Where possible, `bcc`

will place bounds checking into such empty slots. Because `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 does not automatically handle statically allocated global arrays. We have modified the benchmarks by hand to transform such arrays into dynamically allocated arrays, which 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.

ICRs can also be used to check array bounds for Java applications. However, the application of ICRs to Java arrays is slightly different than C. In the next section, these differences are described. Then, modifications to GNU Java compiler and runtime libraries in order to perform implicit bounds checking with ICRs is detailed.

3.4 ICRs FOR JAVA

The general concept of the ICR is no different for Java than C. Any reference to the array must be confined within the ICR. However, Java semantics impose various restrictions that allow Java compilers to perform two bounds checks with a single index expression comparison. Therefore, an n dimensional Java array requires n checks per access instead of $2n$. Fortunately, the same semantic rule removes the need to perform the low bound check for the last dimension, as is required with ICRs for C. Thus, the number of required bounds checks for an n dimensional array access is 0, rather than n.

Consider an example similar to that described above. Place a one-dimensional integer array, indexed only by *positive* 32-bit expressions, at the *beginning* (not the center) an ICR of size 16GB. It is then impossible to generate a reference outside the bounds of the ICR. This is due to three factors. First, arrays in Java are limited to $2^{31} - 1$ entries by the language specification. Second, negative index expressions are not permitted by Java. Third, `gcj` (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

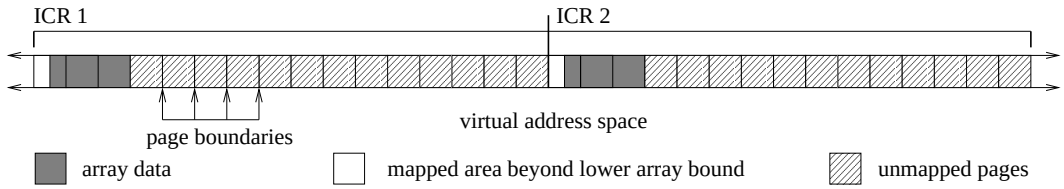

Figure 3.2: Two index confinement regions for Java arrays. Each Java array is placed at the beginning of the ICR and is isolated from any other program data; bounds checking is done automatically through unmapped pages.

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.

This size of each ICR is computed in the same manner as described for the C implementation. The size is a multiple of 4GB (2^{32}) and the size of the element type. Figure 3.2 shows two regions and their associated Java arrays. We align arrays so that the upper limit of the array is at a page boundary, leaving the bottom end of the array unaligned. As before, this allows implicit 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 optimization 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.

3.5 GNU JAVA IMPLEMENTATION

We created a new Java implementation, based on `gcj`, that makes use of ICRs. We modified both the compiler and the runtime libraries; the modifications are described in turn below.

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 indexing expression, as this might result in an access to a mapped portion of a different ICR (see Figure 3.3). 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, it is zero-extended. 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 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. These methods extend naturally to n-dimensional arrays, as the first $n - 1$ dimensions are arrays of objects. The keyword `new` invokes `NewMultiArray`, which calls itself

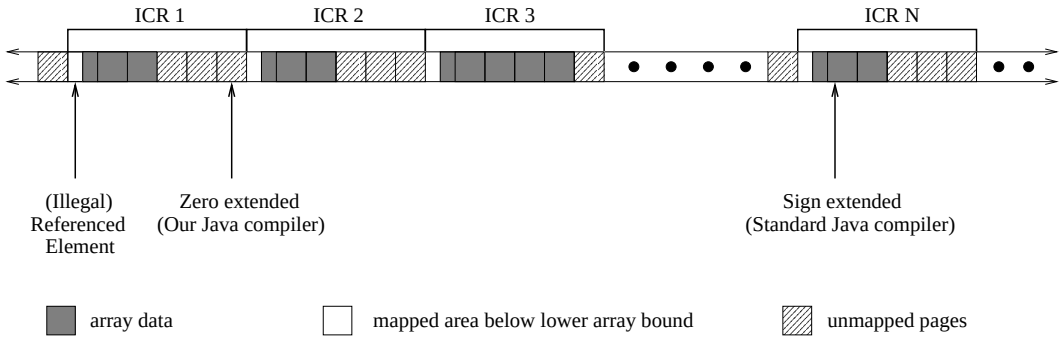

Figure 3.3: Pictorial description of an illegal array reference and its corresponding sign-extended index expressions versus zero-extended index expressions with ICRs.

recursively, calling `NewObjectArray` to allocate each row in the first $n-1$ dimensions. Finally, `NewMultiArray` calls `NewPrimArray` to allocate the last dimension if it is primitive; otherwise, `NewObjectArray` is used to allocate the last dimension. 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.

Our ICR abstraction places significant pressure on the memory hierarchy, including the cache, TLB, and main memory. The cache problems are primarily in cache conflicts, as ICRs are page aligned, and therefore small arrays in consecutive ICRs tend to use the same cache index sets. The TLB and main memory problems are primarily due to internal fragmentation we add; memory allocators generally pack data onto pages. Smaller page sizes lessen this problem. A more subtle problem is that the Linux kernel uses a three-level linear page table, which results in significant amounts of memory consumed by page tables when using ICRs. The next section describes how we modified the kernel to obtain a large address space using a small page size as well as how to reduce memory costs due to page tables.

3.6 LINUX SUPPORT FOR ICRS

Current implementations of IA-64 Linux present several obstacles for the ICR scheme described above. First, programs that use ICRs require a large address space and, for efficiency, a small page size. The former is necessary because ICRs must be allocated a large distance from one another. The latter is beneficial because a small page size reduces internal fragmentation and makes better use of the cache. Unfortunately, with the standard Linux kernel, address space size increases with the page size. For example, 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. Second, we discovered when running some benchmarks (e.g., Multigrid from the NAS benchmark suite) that we quickly caused thrashing in the memory system due to the large number of page table entries that are stored in physical memory.

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 customizable virtual addressing scheme to reduce memory consumed by page tables.

The rest of this section is organized as follows. First, we describe the current IA-64 Linux address translation scheme. Then, we describe the implementation of the two components of the *xvm* abstraction: extending the address space with small pages and customizing the virtual addressing scheme.

3.6.1 STANDARD LINUX VIRTUAL ADDRESS SCHEME

On the IA-64, virtual addresses are first presented to the TLB. If a translation is found, then the physical page is accessed. If a translation is not found, a TLB miss occurs, and there is a hardware lookup of this page in the Virtual Hash Page Table (VHPT) walker. If this lookup fails, the OS is invoked to handle the page fault. Linux uses a three-level page table (PT) for translation. Borrowing Linux terminology, we refer to a page table as a *directory*. Specifically, we call the the top level of the PT the first-level page directory (L1PD); each entry in the L1PD points to a second-level page

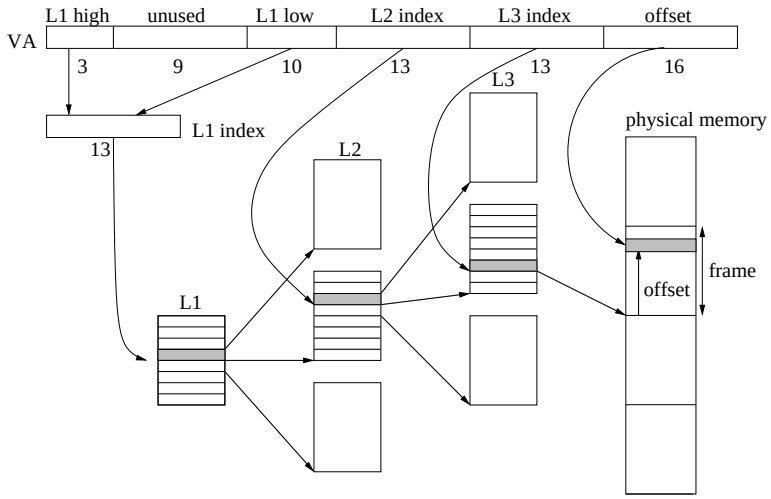

Figure 3.4: Address space layout and address translation for regular processes with a three-level page table in Linux, using 4KB pages. This picture is largely borrowed from [16].

directory (L2PD). Similarly, each entry in the L2PD points to a third-level page directory (L3PD). Finally, the L3PD actually contains entries that map the virtual page to a physical frame. It is important to note that each directory itself is a page in size and is allocated on demand. Figure 3.4 shows the translation scheme pictorially.

3.6.2 Increasing the virtual address space for small pages

The first part of implementing an *xvm* is to increase the address space size. We modified the 4KB kernel directory structure to allow the large virtual address space allowed by the 64KB kernel, while maintaining the cache and memory benefits of the 4KB page size. Simply stated, we modified the directories to be held in several consecutive pages. This allows a large virtual address space with 4KB pages[2].

However, to do this, the implementation of the virtual address scheme had to be changed from that of a standard 4KB page kernel, in which there are 3 directory indices in a virtual address (with

[2]While it is true that some applications, such as databases, desire a 64KB page, we argue that multiple page sizes offered by modern architectures will allow ICRs to use a small page size and other applications to use a large page size. Section 5 discusses this further.

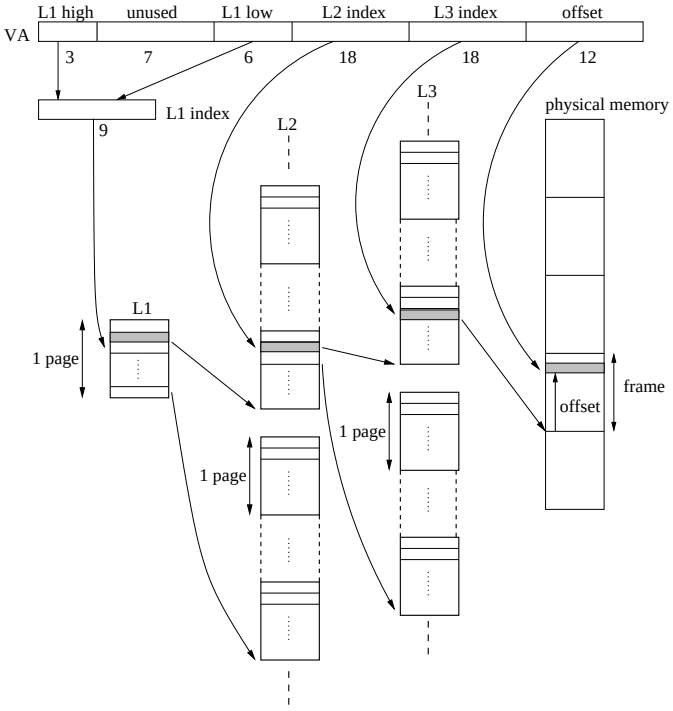

Figure 3.5: Address space layout for processes using our *xvm* abstraction. The L2PD and L3PD are each 512 pages instead of 1 page.

9 bits each) that are used to index the first, second, and third levels of the page table. For simplicity, we left the L1PD at 1 page in size and so still need only 9 bits for its index. Any directory that is built of N consecutive pages, needs $log_2 N$ additional bits for the index at that level. For example, increasing the second and third level directories 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 3.5 shows the new scheme. The total number of bits used in a virtual address is 57, which is necessary to allocate 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, and the maximum additional memory usage due to the L2PD is 128MB—which only occurs if all of virtual memory is used[3].

[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.

The drawback of using multipage directories is that the minimum size of a process increases. For example, if 512 pages are used for the L2PD and L3PD, the minimum size of a process grows from several kilobytes to several megabytes. This is clearly unacceptable, as a large number of system processes will occupy much of physical memory (as directories are stored in memory; see below).

Hence, a process uses an *xvm* through a new system call `enable_xvm`, which converts a standard Linux 3-level page table (as shown in Figure 3.4) into an multipage, multilevel extended version (as shown in Figure 3.5). This allows only those processes that make use of ICRs to incur the memory overhead due to the larger directories when using *xvm*. The only effect on regular (non-ICR) processes is that somewhat less physical memory is available, due to processes that use ICRs and *xvm* directories; this could be alleviated if Linux supported swapping of directories.

The implementation of `enable_xvm` is as follows. First, a new page table structure with multilevel, multipage directories is created. Second, the current page table is copied into the new page table. Third, all page table indexing routines are changed. In practice, the overhead for `enable_xvm` is negligible, because in our benchmarks it is invoked at the start of the program, which means only tens of directories are copied.

Linux uses demand paging for virtual memory pages as well as for directories themselves. However, directories reside in physical memory from their allocation time until process termination. Thus, a large number of directory allocations (which can occur with ICRs) can consume significant amounts of memory, leaving little for actual program data. We discuss our solution to this problem, which is to customize virtual addressing, next.

3.6.3 CUSTOMIZED VIRTUAL ADDRESS SCHEME

As discussed above, ICRs violate the principle of locality. For example, a 4KB page size contains 512 entries, so in an extended virtual address space as described above, each L3PD contains 256KB entries, assumed to represent *contiguous* virtual memory. Hence, one L3PD represents 1GB (256KB $\times$ 4KB) of virtual memory. Unfortunately, using the scheme shown in Figure 3.5,

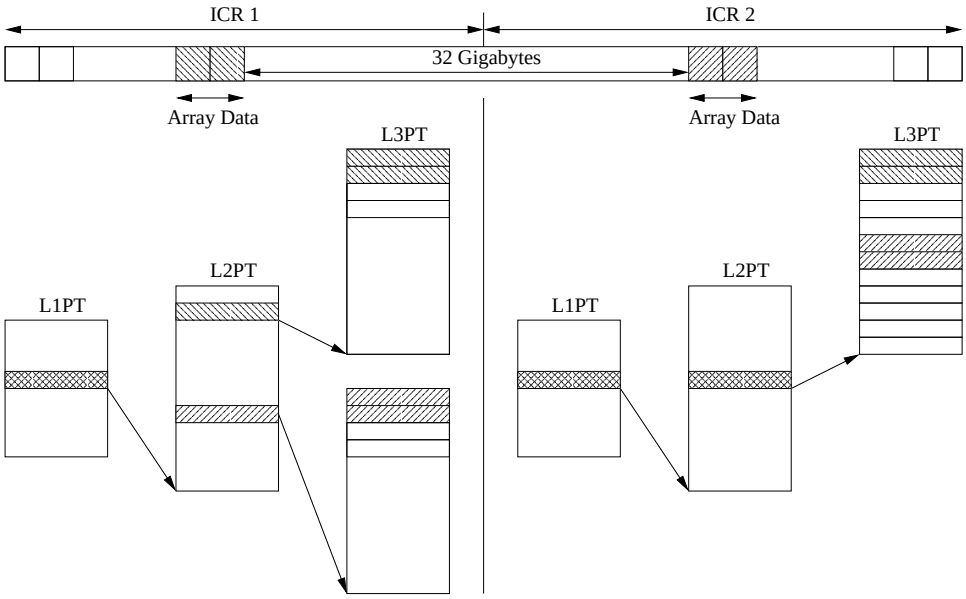

Figure 3.6: The original (left) and *xvm* (right) page table indexing schemes. The fill patterns shown in the directories correspond to the mapped pages in the ICRs.

arrays placed in ICRs are at least 16GB from one another, meaning that two different array references require allocation of *two* L3PDs—even though typically only *one* entry in each L3PD will be used. This is shown on the left-hand side of Figure 3.6. A similar problem occurs with the L2PDs, although to a lesser extent. As a result, memory consumption is increased considerably. This is a well-known problem with extremely sparse address spaces, which are in general better served using a hashed or clustered page table [17].

However, ICRs are predictable distances from one another. In other words, they exhibit regular sparse patterns; this is especially true for multidimensional arrays, where a given dimension has vectors with identical sizes and types. We can take advantage of this to customize an indexing scheme for ICRs. Our solution is to break up the L2PD and L3PD indices into two components and select bits for them based on the expected sparse access pattern for ICRs. The right-hand side of Figure 3.6 shows our scheme pictorially. We accomplish 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.

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; then, the routine can 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 lookups in hardware, 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 the 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.

CHAPTER 4

EXPERIMENTAL RESULTS

We examined the performance of array bounds checking and ICRs on several C and Java benchmarks. These performance results indicate that compiler bounds checking can increase the execution time of programs by as much as a factor of 2. In addition, ICRs perform better that compiler bounds checking for all of the C and Java benchmarks.

We performed our experiments on a 900MHz Itanium 2 processor with 1.5GB main memory, a 16KB L1 instruction cache, 16KB L1 data cache, 256KB L2 cache, and 1.5MB L3 cache. All three levels of the cache are on chip. 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 as follows. First, we measure the abilities ABCD [1], a state-of-the-art Java static bounds checking elimination scheme. Then, we discuss ICR performance results for C benchmarks, and we examine low level performance monitors to understand the overhead introduced by bounds checking and ICRs. Finally, we present performance results for our Java benchmarks and examine the performance monitors.

4.1 STATIC REMOVAL OF BOUNDS CHECKS

We studied a subset of our Java benchmarks[1], which are described in Section 4.3, to determine how many bounds checks could be eliminated statically by ABCD [1] (see Table 4.1). First, we had to modify ABCD to handle integer constants in array expressions, because the unmodified ABCD removed no bounds checks in even the simplest programs[2]. It is important to note that this modification makes ABCD overly optimistic in that it removes some checks that are not possible to

[1]The larger programs were difficult to study with ABCD.

[2]The ABCD `README` file states that integer constants are not yet handled.

Program	*NAS/Hand-written*	*Percentage of Checks Eliminated (Modified/Not Modified)*
FT	NAS	0% / 0%
LU	NAS	0% / 0%
MG	NAS	0% / 34.9%
BT	NAS	0% / 0%
MM	Hand-written	100% / 0%
JAC	Hand-written	100 % / 36%
TOM	Hand-written	100 % / 54%
MG3D	NAS (using modified `f2java`)	70 % / 0%

Table 4.1: Percentage of bounds checks, for a subset of our Java benchmarks, eliminated by ABCD for the 5 most heavily executed functions in each program. The benchmarks are either from the NAS suite or hand written, as indicated. They are described in Section 4.3. *Modified* means that we changed ABCD to recognize integer constants, even though this means the analysis was not strictly correct.

prove statically—our modified ABCD represents an upper bound on how many checks ABCD can remove. For each program, we investigated the percentage of bounds checks eliminated in the five functions that consumed the largest percentage of execution time. As can be seen from Table 4.1, ABCD performs well (with our modification) on the hand-written benchmarks (MM, JAC, TOM), which are relatively simple. However, on the more complicated NAS programs, there are a significant number of array references that ABCD cannot prove are within their bounds. Furthermore, on MG3D, the 70% number is misleading, because ABCD incorrectly removed many of the checks (due to our modification). Our manual inspection revealed that some of the references for which ABCD can not prove anything are actually theoretically removable. However, though possible, it would often be impractical. We believe that for realistic programs, which often contain complicated array indexing, pointers, and/or multiple call sites with different-sized array parameters, static elimination of bounds checks is unlikely to eliminate most or all of the necessary checks.

Rather than removing bounds checks statically, we use ICRs to implicitly check array bounds at runtime with ICRs. In the next section, we present the performance results for ICRs with C benchmarks.

4.2 C EXPERIMENTAL RESULTS

We examined the performance of array bounds checking on several C programs from the NAS benchmark Suite 2.3 [2], plus a few additional programs. The NAS programs used include **CG**, a conjugate gradient program; **EP**, an embarrassingly parallel program (we only use one processor); **FT**, a Fourier transform; **IS**, an integer sorting program; **LU**, LU decomposition, **MG**, a multigrid solver, and **MM**, matrix multiplication. We had to hand-write LU, because `bcc` mishandles pointer arithmetic in the NAS versions and hence does not produce correct output[3]. (We are working to fix these `bcc` bugs.) Our other programs include **JAC**, a hand-written version of Jacobi iteration, and **TOM**, which is the TomcatV mesh generation program from SPEC92. The NAS suite was run with Class W input size and the hand-written benchmarks were each run with input size 512×512.

We studied each of our benchmarks to determine how many bounds checks could be eliminated statically. Our study showed that CG, MG, and FT contain many array references that are very difficult to prove are within their bounds. Even if it were possible, it would be impractical. The other programs contained array references that could conceivably be removed by an optimizing compiler. However, we tested the entire Java NAS suite using ABCD [1], a state-of-the-art Java static bounds elimination scheme. In each program in the NAS suite, ABCD was unable to remove many checks in time-consuming loops. This is discussed further in Section 4.1.

All multidimensional arrays in all C benchmarks are allocated using a vector of vectors style, which is required for ICRs. While this allocation style results in slightly more overhead to access array elements than a contiguous multidimensional array, separate tests showed that this overhead was negligible in our benchmarks.

The rest of this section first discusses the overall execution times of several programs. Then, we further examine some of the results through inspection of hardware-level counters.

[3]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.

4.2.1 Overall Execution Times

Figure 4.1 shows the cost of several versions of each C program. Our baseline program (which we denote *No Checks*) has no bounds checks, the VHPT walker enabled, and does not use *xvm*. It is this program that all of the other programs are normalized against. We used a kernel with 8KB pages for the baseline programs, because that was the page size that performed best among 4KB, 8KB, 16KB, and 64KB[4]. *Full Checks* refers to the program that has $2n$ bounds checks for each n-dimensional array reference. This program is generated by `bcc`; it also has the VHPT walker enabled and does not use *xvm*. *ICR No Checks* measures the overhead of ICRs; it is the same as *No Checks* except that arrays are allocated in ICRs (no bounds checking is done), the VHPT walker is disabled, and *xvm* is used. Our technique, which we denote *ICR One Check*, is the same as *ICR No Checks*, except that a low bounds check is added in the last dimension. Finally, *ICR Two Checks* adds a low *and* high bounds check in the last dimension (but allocates this dimension contiguously via `malloc` rather than in ICRs). Note that *ICR No Checks* and *ICR Two Checks* are included so that we can compare them to our method (*ICR One Check*).

Figure 4.1 shows execution times of all of our C benchmarks for each program version. All benchmarks run faster when using *ICR One Check* than when using *Full Checks*. Overall, *ICR One Check* 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 branch. Inspection of *ICR No Checks* also shows that the overhead of *ICR One Check* is in general fairly evenly divided between the cost of ICRs and the cost of the single bounds check. In the particular case of FT, *ICR One Check* has an overhead of 88%, compared to 96% for *Full Checks*. This is close to a worst-case scenario for ICRs and is described further in Section 4.2.2 with the help of performance counters.

Figure 4.2 displays the benefit of using a small page size. It shows that the performance of many of our benchmarks (for *ICR One Check*) using a 4KB page size is similar to the performance when using a 16KB or 64KB page size. However, for programs that have extremely high rates of internal fragmentation (FT and MG), the improvement with 4KB pages is significant. The reason

[4]While the number of TLB misses is smallest with the 64KB kernel, the performance of the 8KB kernel was slightly better.

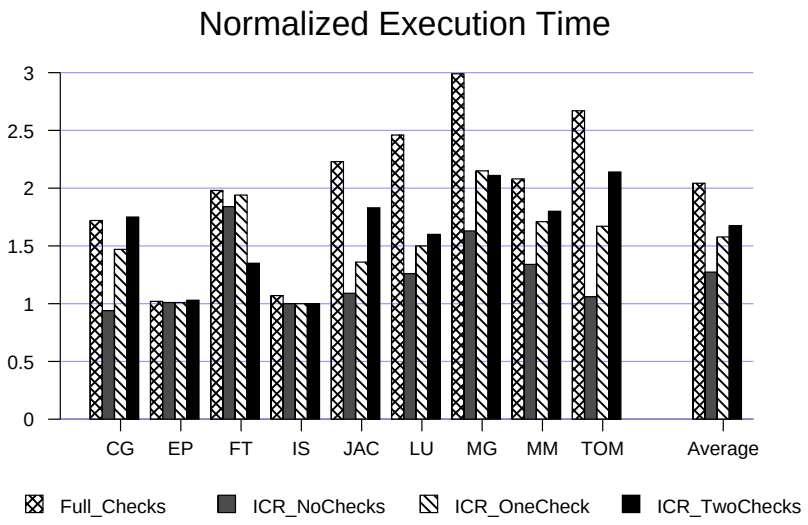

Figure 4.1: Execution times for each program version on each C benchmark. All times are normalized to the *No Checks* version, as it is the baseline; this means that smaller bars are better. Using *ICR One Check* is better than *Full Checks* in all programs. Note that the benchmarks are explained at the beginning of Section 4.

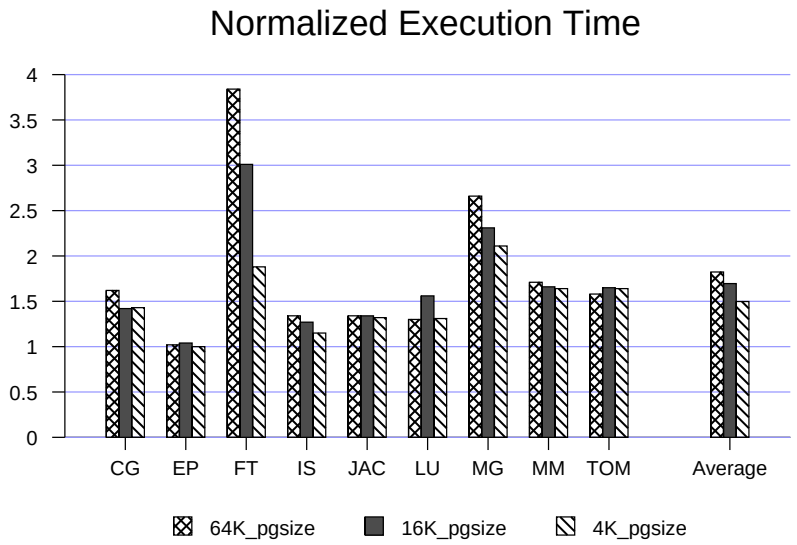

Figure 4.2: Execution times for *ICR One Check* on each C benchmark across different page sizes. All times are normalized to the *No Checks* version, so this means that smaller bars are better. The 4KB page size allows much better performance than 16KB or 64KB.

for this is that there are fewer cache conflicts with a 4KB page size than with 16KB or 64KB. It is also possible to run programs that use about four times as much memory when using ICRs and a 4KB page size.

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. Our investigation showed that the problem is not with `bcc`, as its expectation is that the `gcc` back-end should perform all code motion. Instead, the problem is simply that `gcc`, a state-of-the-art compiler, cannot hoist any checks in our benchmarks—a reminder that while algorithms for code motion are mature, in practice it is hard to legally move code without risking modification of program semantics.

To investigate the improvement in *Full Checks* if checks were to be hoisted, we manually removed all but the last-dimension checks from *Full Checks*. We found that even when removing these checks, only in FT was *Full Checks* better than *ICR One Check*. Note that this represents an *optimistic* view of *Full Checks*, as even if checks are hoisted, they still must be executed in outer loops.

4.2.2 Low-Level Performance Details

The performance of *ICR One Check* is better than that of *Full Checks* in all of our benchmarks. However, the absolute performance of *ICR One Check* on two benchmarks, FT and MG, is a factor of two worse than using *No Checks*. We further examined the cause of this using hardware performance counters. We determined that the primary cause of the overhead was due to large increases in the numbers of both L3 cache misses and TLB misses.

The left half of Figure 4.3 shows the number of L3 cache misses for *ICR One Check* on FT and MG using a 64KB, 16KB, and 4KB kernel, normalized to the number of L3 cache misses for *No Checks*. Two things are evident. First, the number of L3 misses is significant, as there are up to 50% with *ICR One Check* than with *No Checks* for FT. This is a large part of the reason why FT

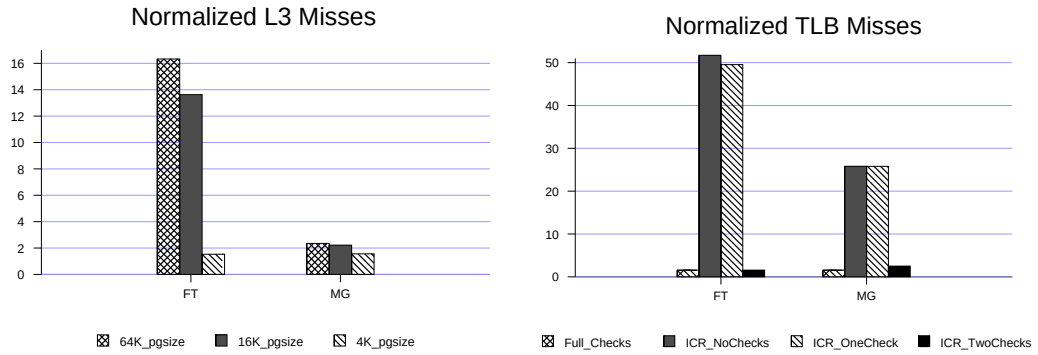

Figure 4.3: Low-level performance counter results for L3 cache misses (left) and TLB misses (right) for FT and MG. The L3 misses graph shows results for *ICR One Check* across different page sizes, whereas the TLB misses graph shows results for *Full Checks*, *ICR No Checks*, *ICR One Check*, and *ICR Two Checks* using a 4KB page size. All times are normalized to the *No Checks* version.

performs poorly using *ICR One Check*. (Keep in mind, however, that despite this, *ICR One Check* still outperforms *Full Checks*.) Second, by using a 4KB kernel allowed by the *xvm* abstraction, we are able to greatly reduce the number of L3 misses (by up to an order of magnitude) compared to a kernel using 64KB or 16KB pages. L2 cache misses are also reduced, but to a lesser extent.

The right half of Figure 4.3 shows the number TLB misses (only for the 4KB kernel) for the different program versions. As expected, the degradation of the TLB performance when using ICRs is dependent on array sizes, particularly the size of the last dimension, and is the other component causing the relatively poor performance with ICRs for FT and MG.

In general, we see a trade-off between *ICR One Check* 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. In our tests, the trade-off always favors *ICR One Check*. We see that with FT in particular, which is the worst case in our benchmarks for *ICR One Check*, the main array in FT is $128 \times 128 \times 32$. This means that with *ICR One Check*, there will be more than 16,000 ICRs. Combined with the fact that there is not much computation per array reference, the performance of *ICR One Check* is dominated by TLB misses. Note that for row sizes that are

this small, another option is to use ICRs for the first $n-1$ dimensions only, and allocate the last dimension contiguously; this requires two bounds checks in the last dimension. The performance of *ICR Two Checks* is better than that of *ICR One Check* for both FT and MG. However, *ICR One Check* is superior for all of the other programs. In both cases, the low computation per last dimension row makes the overhead of TLB misses nearly as high as that of full bounds checking, but by packing the last dimension of the array, we avoid most of this overhead. The trade-off between *ICR One Check* and *ICR Two Checks* is described further in Section 5.

4.3 JAVA 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 [18], 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**, LU decomposition; **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 bounds checking scales with dimensionality. We used maximum optimization (-O6)

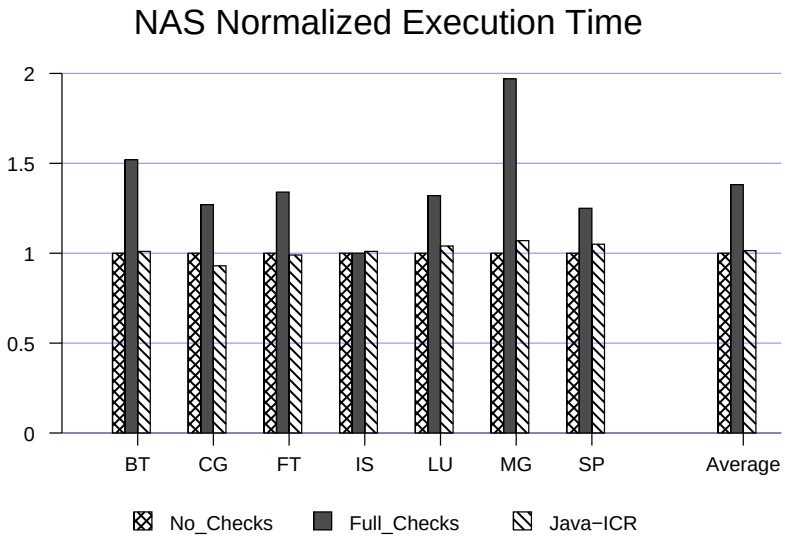

Figure 4.4: Execution times for each program version on each of the NAS benchmarks (using linearized arrays). All times are normalized to the *No Checks* version, as it is the baseline; this means that smaller bars are better. Using *Java ICRs* is better than *Full Checks* in all programs. Note that the benchmarks are explained at the beginning of Section 4.

to compile our programs with both Java implementations. In addition, we compiled all benchmarks directly to executable programs rather than Java bytecodes.

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

4.3.1 Overall Execution Times

Figure 4.4 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 bound check for each dimension. As described in

Section 3.4, this is due to the restriction that indices cannot be negative, allowing the compiler to compare the index to only the upper bound of the array. 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 50%, while the average overhead for *Java ICRs* is only 3%. 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 [18]. 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 is due to fewer bounds checks (only one check is needed per array access), and the latter is due to 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 another dimension. 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[5] 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.

We are not yet done translating the other benchmarks. This is because the translation is a time-consuming task, as a straight translation using `f2java` does not produce completely correct code.

[5]The original `f2java` linearizes arrays.

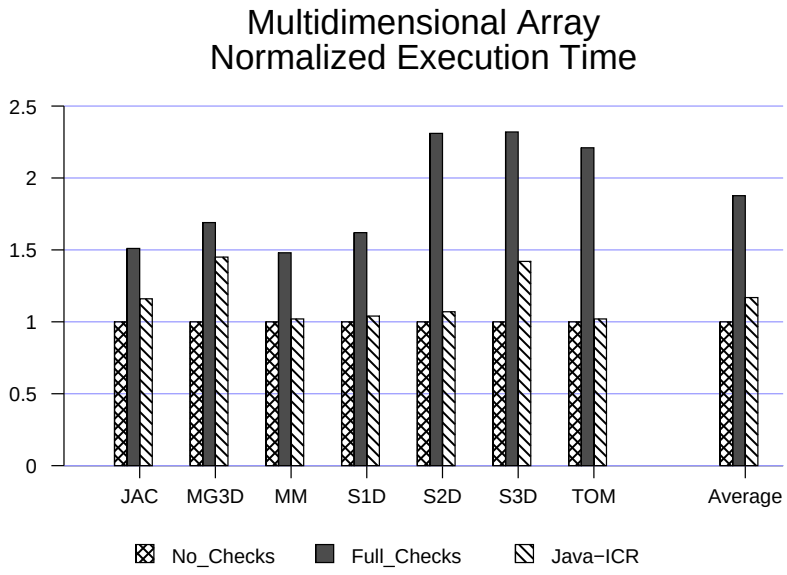

Figure 4.5: Execution times for each program version on each of the hand-written benchmarks as well as MG3D. All times are normalized to the *No Checks* version, as it is the baseline; this means that smaller bars are better. Using *Java ICRs* is better than *Full Checks* in all programs. Note that the benchmarks are explained at the beginning of Section 4.

Hence, we also tested several multidimensional programs written by hand: MM, JAC, TOM, and the three synthetic programs.

Figure 4.5 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 *Java ICRs* overhead increase 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, and *Java ICRs* still is 24% better than *Full Checks*. 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).

This is strong evidence that the de-linearized NAS suite will perform much better with *Java ICRs* than with *Full Checks*. Combined with the results from Section 4.2, which show that the multidimensional C versions of these programs perform well with ICRs relative to explicit bounds checks, we believe that the NAS multidimensional Java programs will also perform well.

As previously stated, 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. Since `gcj` uses the same optimizing backend `gcc` that `bcc` uses, we cannot expect any hoisting of bounds checks from the innermost loops, even when it is theoretically, although not practically possible.

4.3.2 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 4.6). 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. Despite the increase, the actual execution time of the bounds checking program does not increase by such a large amount. This is explained by examining the number of `nop` instructions executed. Recall that the Itanium executes instructions in bundles, which often contain `nop` slots. In many cases, the original code exhibited poor ILP, allowing bounds checking to be folded into the `nop`. However, this was not always the case, so *Full Checks* still pays a heavy penalty for TOM. Second, the *Full Checks*

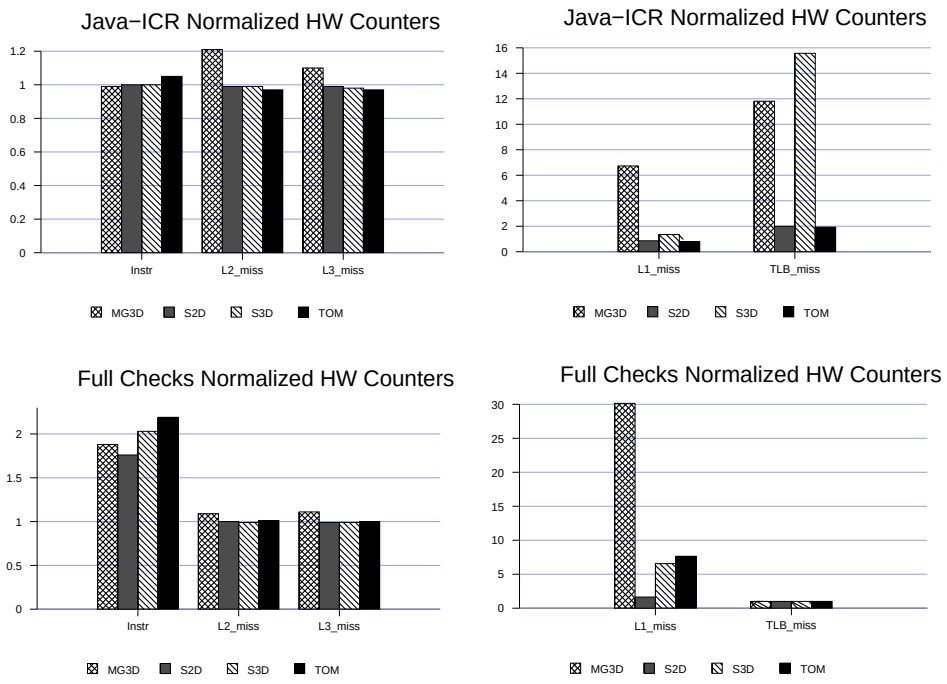

Figure 4.6: Low-level performance counter results for *Full Checks* and *Java ICRs* for TOM, S2D, S3D, and MG3D. All times are normalized to the *No Checks* version.

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 4.6 is the large increase in TLB misses for S3D and MG3D. 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 the L1 cache for MG3D, mostly due to the use of a 4KB page—

experiments presented in Section 4.2 revealed that using a 16KB or 64KB page size caused a large increase in L3 cache misses when using ICRs. Both of these aspects are shown in Figure 4.6.

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 4.6) 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 trade-off between our scheme using ICRs and those that use full compiler checking; 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, *ICR One Check* for C arrays and *Java ICRs* for Java arrays significantly reduce the average penalty for performing bounds checks in C and Java, respectively. Our experiments show that *ICR One Check* reduces the average penalty for bounds checking in C from approximately 100% to nearly 50%. Also, *Java ICRs* can reduce the average overhead of bounds checking in Java from 63% to 9%.

CHAPTER 5

FUTURE WORK

Our results show that the ICR abstraction is effective in greatly reducing the cost of checking array bounds. This section presents several issues that arise with ICRs that are promising directions for future work. These issues include page size, selective use of ICRs, and speculation.

5.1 PAGE SIZE

As was discussed in Section 3, ICRs benefit from the use of a small page size. However, other applications may desire large page sizes. Fortunately, 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 [19]. Combined with our *xvm*, we argue that ICRs can use small page sizes with little, if any, effect on other applications.

5.2 SELECTIVE USE OF ICRS

A future direction would be to integrate our technique with static analysis techniques for removing array bounds checks. This means that we would only place arrays that might be accessed in a potentially unsafe way into ICRs. Arrays that are always (provably) accessed within bounds would suffer no performance penalty for bounds checking. Also, if too much memory is used by ICRs, we can use traditional compiler techniques to estimate the most frequently accessed arrays and place only those arrays into ICRs; other (less frequently accessed) arrays could be bounds checked in the traditional manner. This would offer a performance improvement over checking all arrays.

We also would like to selectively use ICRs when either (1) it is better to use ICRs with two checks (*ICR Two Checks*) than *ICR One Check*, or (2) it is better to use full bounds checking (*Full*

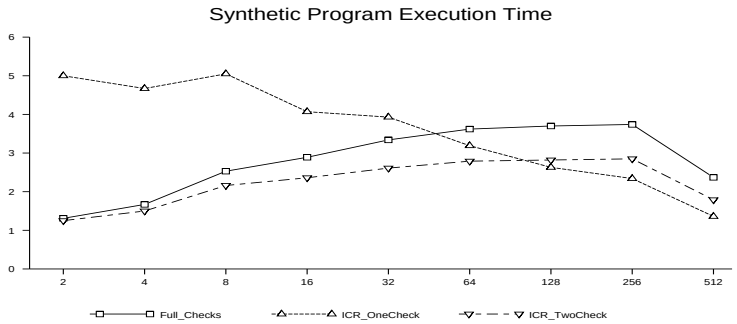

Figure 5.1: Execution times for the synthetic benchmark with final dimension varying from 2 (the worst case) to 512. All times are normalized to the *No Checks* version.

Checks) than *ICR One Check*. We investigated the break-even points for both of these cases using a synthetic application that creates a $128 \times 128 \times D$ array of integers. Each element is updated with a single store; this is repeated many times. We varied D from 2 to 512, testing *No Checks*, *ICR One Check*, and *ICR Two Checks*. The results are shown in Figure 5.1. We found that the break-even point between *ICR One Check* and bounds checking occurred when D was between 32 and 64. Furthermore, this point for *ICR One Check* vs *ICR Two Checks* was between 64 and 128. Finally, *ICR Two Checks* is *always* better than *Full Checks*; this means that for our programs we can always improve performance by using *ICR Two Checks*. Hence, it is important to find ways to determine whether to use *ICR One Check* or *ICR Two Checks*; it may be possible to use static analysis for this purpose.

5.3 SPECULATIVE INSTRUCTIONS

The ICR technique increases the number of TLB misses, as more pages are used than would be used with typical memory allocation techniques. One way to decrease the impact of these misses is through the use of speculative and prefetching instructions [20]. The IA-64 architecture provides several such instructions. We could extend the compiler to insert explicit prefetching code or speculative loads, which allows the servicing of the TLB miss without blocking. Preliminary

experiments indicate that this can result in a noticeable performance improvement (approximately 10% for a synthetic benchmark).

CHAPTER 6

CONCLUSION

We have introduced a new abstraction called index confinement regions (ICRs), which are isolated virtual memory regions. ICRs allow complete implicit bounds checking for C and Java arrays without much of the runtime penalty associated with explicit compiler bounds checking. For C programs, ICRs allow implicit bounds checking for the first $n-1$ dimensions of an n dimensional array and require one check in the last dimension. The resulting reduction from $2n$ checks to 1 allows our C benchmark programs using ICRs to execute faster in all cases than the corresponding program with full bounds checking. In particular, the average improvement is close to 50%. For Java programs, ICRs provide complete implicit bounds checking for all n dimensions of an n-dimensional array with zero explicit bounds checks. This reduction from n explicit checks to zero results in a bounds checking overhead of only 9% in our benchmarks—a reduction of 54%.

In order to obtain this improvement, it was necessary to (1) use a small (4KB) page size with a large virtual address space as well as (2) optimize sparse ICR access patterns. 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 achieved these two steps through an abstraction we called *xvm*, which provides an extended, customizable virtual memory, with little, if any, effect on other processes. Overall, we believe that ICRs along with *xvm* are a promising technique for reducing array bounds checking overhead.

BIBLIOGRAPHY

[1] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *SIGPLAN Conference on Prog. Language Design and Implementation*, pages 321–333, 2000.

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

[3] Greg McGary. Bounds-checking C compiler (http://www.gnu.org/software/gcc/projects/bp/main.html).

[4] Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. In *SIGPLAN Conference on Prog. Language Design and Implementation*, pages 270–278, 1995.

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

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

[7] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *SIGPLAN Conference on Prog. Language Design and Implementation*, pages 249–257, 1998.

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

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

[10] Tzi cker Chieuh. Personal communication. October 2002.

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

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

[13] Christopher Bentley, David Lowenthal, and Scott Watterson. Operating system support for low-cost array bounds checking on 64-bit architectures.

[14] Michael Young, Jr. Avadis Tevanian, Richard Rashid, David Golub Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *11th Symposium on Operating Systems Principles*, November 1987.

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

[16] David Mosberger and Stephane Eranian. *IA-64 Linux Kernel: Design and Implementation*. Prentice-Hall, 2002.

[17] Madhusudhan Talluri, Mark D. Hill, and Yousef A. Khalidi. A new page table for 64-bit address spaces. In *Symposium on Operating Systems Principles*, December 1995.

[18] Michael Frumkin, Matthew Schultz, Haoqiang Jin, and Jerry Yan. Implementation of the NAS parallel benchmarks in Java. NAS-02-009, NASA Ames Research Center.

[19] Simon Winwood, Yefim Shuf, and Hubertus Franke. Multiple page size support in the Linux kernel (http://www.cse.unsw.edu.au/ sjw/linux-mpss/slides/img0.htm).

[20] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher J. Hughes, Yong Fong Lee, Dan Lavery, and John P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *28th International Symposium on Computer Architecture*, pages 14–25, July 2001.