A Comparison of Array Bounds Checking on Superscalar and VLIW Architectures*

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

Abstract

Several programming languages guarantee that array subscripts are checked to ensure they are within the array bounds. While this guarantee improves the correctness and security of array-based code, it adds overhead to array references. This is unacceptable in array-intensive scientific codes; some estimates show more than 60% overhead for array bounds checking. This performance limitation is a significant obstacle preventing the scientific community from adopting compiler-enforced array bounds checks (e.g., Fortran, if bounds checks are enabled, or Java). Previous research has explored optimizations to statically eliminate bounds checks, but the dynamic nature of many scientific codes makes this difficult.

We are interested in techniques for reducing the cost of array bounds checks, with the goal of improving the performance of scientific codes. In this paper we evaluate the potential gain of such techniques by analyzing several scientific benchmarks on two different types of architectures. We performed experiments on a Pentium 3, Pentium 4, and an Itanium 1; results show that bounds checking on the Pentium 4 has moderate overhead, averaging 34%. The Itanium, however, incurs much more significant overhead, averaging 74%. Finally, the Pentium 3 has the largest overhead, at 117%. For this paper we added bounds checks to C through `bcc`, a bounds checking extension to `gcc`. We expect array bounds checking overhead to be similar in other languages.

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. For this reason, languages such as C do not require checking of array bounds.

Despite this overhead, array bounds checks are important for two reasons. First, array access violations are a frequent source of error and are particularly difficult to detect and fix. These

*This research was supported in part by a State of Georgia *Yamacraw* grant.

violations may cause errors in seemingly unrelated parts of the code and often occur only under exception conditions. For this reason, some languages (e.g., Java) require array bounds checking. Second, checking bounds can close security loopholes such as stack smashing [1]. Some estimates indicate that buffer overruns account for more than 50% of security vulnerabilities [2].

While array accesses are infrequent in many 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. 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.

Hence, our goal is to provide a combination of static and dynamic techniques to provide *low-overhead* array bounds checking for scientific applications. In this abstract, we wish to evaluate the potential benefit we can achieve if our project is successful. We study the performance of several scientific benchmarks with array bounds checking provided by `bcc`, a state-of-the-art bounds checking extension to `gcc`. We evaluate performance on both a Pentium 3 and a Pentium 4, which are superscalar architectures, and an Itanium 1, which is a very long instruction word (VLIW) architecture.

Our results show that array bounds checking has large overhead on the Pentium 3 and Itanium 1, averaging 117% and 74%, respectively. On the Pentium 4, the overhead is a more moderate 34%. This suggests that significant benefit could be achieved on the Itanium by partially or completely eliminating bounds checks.

The remainder of this abstract is organized as follows. The next section briefly describes related work. Section 3 provides implementation details, and Section 4 presents performance results. Finally, Section 5 concludes and discusses future work.

2 Related Work

A significant body of work exists on static analysis to eliminate array bounds checks. Kolte and Wolfe [3] perform partial redundancy analysis to hoist array bound checks outside of loops. Their algorithm was based on that described by Gupta [4], who formulated the problem as a dataflow analysis. Bodik al. [5] 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; it works for both statically and dynamically allocated regions.

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 scientific applications where indices cannot be determined at compile time. As one example, our inspection of the FT benchmark from the NAS suite shows that it is extremely difficult, if not impossible, to prove that array references are within bounds. In these

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

Figure 1: Representation of pointers in bcc.

```
int *A = malloc(10 * sizeof(int));
void foo() {
  A[5] = 999;
}
```

Figure 2: Example C source code.

cases compile-time schemes to eliminate bounds checks fail, and the compiler must fall back to general run-time checking.

Several have studied handling buffer overflow; this includes using a gcc patch along with a *canary* to detect it [10]. Another compile-time solution, RAD [11], involves modifying the compiler to store return addresses in a safe location. This solution retains binary compatibility because stack frames are not modified.

We use bcc [12] for our bounds checking because it adds as little overhead as possible; it stores the bounds along with the pointer. An alternative approach, due to Jones and Kelly [13], is to add extra code to perform a run-time lookup in an associated index structure; advantage is that this approach works with legacy library routines. However, the overhead with this approach is significantly higher than bcc.

3 Implementation

The Bounds Checking Compiler (bcc) is an extension to the GNU C Compiler (gcc). The bcc compiler ensures that any pointer dereference is within the allocated memory for the pointer. To accomplish this, each pointer is assigned a lower and upper bound. These three values are stored in a structure that is associated with the pointer (see Figure 1). The bound data is initialized when memory is allocated for the pointer, statically or dynamically. Dereferencing such a pointer produces several extra assembly instructions that perform bounds checking; the address is compared to the stored lower and upper bounds. If the accessed memory location is outside the bounds, a trap is generated to alert the user of the bounds violation, and the program exits.

Because bcc changes pointer representation, functions that accept a pointer as an argument or return a pointer are prefixed by __BP_. This means that any code linked with bounded pointer programs must be compiled with bcc. For example, the gnu C library, glibc, must have dual library routines, BP and non-BP.

Figure 3 shows the generated code with and without bounds checking on the Pentium 4. Re-

Instruction	Description
movl (A), %ecx	get A
addl 20(%ecx), %edx	get addr. of A[5]
movl $999, 20(%ecx)	A[5] = 999
ret	return

Instruction	Description
movl (A), %ecx	get A
leal 20(%ecx), %edx	get address of A[5]
cmpl (A+4), %edx	compare to low bound
jb L4	jump to trap if $<$ bound
cmpl (A+8), %edx	compare to high bound
jae L4	jump to trap if $>$ bound
movl $999, 20(%ecx)	store 999 into A[5]
ret	return
L4: int $5	generate trap

Figure 3: Example Pentium assembly code both with and without bounds checking.

Instruction 1	Instruction 2	Instruction 3	Description
r10 = @ltoff(A#)	nop	addl r3 = 999, r0	get address of A, set r3 to 999
ld8 r2 = [r10]	ld8 r9 = [r2]	nop	2 indirect loads to get base of A
adds r2 = 40, r9	nop	nop	get address of A[5] into register
st4 [r2] = r3	nop	ret	store into A[5], return

Figure 4: Example Itanium assembly code.

call that A is a structure of three items rather than a pointer to an array. A+4 and A+8 are the locations of the low bound and the high bound, respectively. When static allocation is used, the compiler replaces these locations with constants. In the bounds checking version of the code, four additional instructions are executed, two for each bounds check. Also, the bounds check involves two additional memory references, one for each `cmp` instruction.

Figures 4 and 5 show the generated code without and with bounds checking on the Itanium 1. This code is more complicated to understand due to the *bundles* present on the VLIW-based Itanium. Each (long) instruction contains three regular instructions. In addition, any of the three instructions in a bundle may be *predicated*. This means that it will only be executed if a predicate register is set to true. These predicate registers are set by `cmp` instructions. In the example code shown in Figure 5, both of the array bound checks are implemented with such a sequence. Also note that multiple branch instructions may appear in a single bundle, and predicates may be used to select which (if any) of the branches to execute.

On the Itanium, each bounds check incurs a load, a compare, and a branch instruction. In the example shown in Figure 4, there are `nop` slots in some bundles. However, as can be seen in in Figure 5, scheduling constraints make it impossible to insert all bounds checking instructions into `nop` slots. In the example shown in Figure 5, the bounds checking version typically executes four additional bundles.

Instruction 1	Instruction 2	Instruction 3	Description
r14 = @ltoff(A#)	nop	nop	get address of A
ld8 r15 = [r14]	nop	mov r14 = r15	put address of A into r15 put address in r14 also
adds r16 = 16, r15	ld8 r15 = [r14], 8	nop	set r16 to addr of high bound get pointer into r15 set r14 to low bound
ld8 r14 = [r14]	adds r15 = 40, r15	cmp p6 ← r14,r15	set r14 to low bound set r15 to addr of A[5] check low bound
nop	nop	(p6) branch .L4	goto L4 if violation
ld8 r14 = [r16]	nop	nop	set r16 to high bound
cmp p6 ← r14,r15	addl r14 = 999, r0	(p6) branch .L4	check high bound get 999 into r14 goto L4 if violation
st4 [r15] = r14	nop	ret	store into A[5], return
L4: nop	nop	call abort	call abort

Figure 5: Example Itanium bounds checking assembly code.

4 Performance

We examined the performance of array bounds checking on several programs from the NAS benchmark suite [14], plus a few additional programs. The NAS programs used include CG, a conjugate gradient program; EP, an embarassingly parallel benchmark (we only use one processor); FT, a fourier transform; IS, an integer sorting program, and MG, a multigrid solver. Our other programs include hand-written versions of Jacobi iteration, an iterative relaxation scheme; Tomcatv, a mesh generation program from SPEC 92; matrix multiplication, and LU decomposition. The latter two programs are actually in the NAS suite, but `bcc` produces code that crashes or does not produce correct output. This is likely due to the mishandling of pointer arithmetic by `bcc`. (We are working to remove these bugs.)

These are primarily array-intensive benchmarks, with the majority of execution time spent performing array accesses. Some of these benchmarks use only one-dimensional arrays, while others used two and sometimes three dimensions. As expected, the higher dimensional accesses are more expensive, because two bounds must be checked for each dimension of the array.

As shown in Figure 5, we see that the amount of code that must be executed for array bounds checking can be quite substantial. If both the high and low bounds must be checked, we must execute at least 2 comparisons and 2 branches for each checked array access. Often, there are also extra loads involved.

The remainder of this section describes our experimental setup and the performance results of adding array bounds checks to our benchmarks.

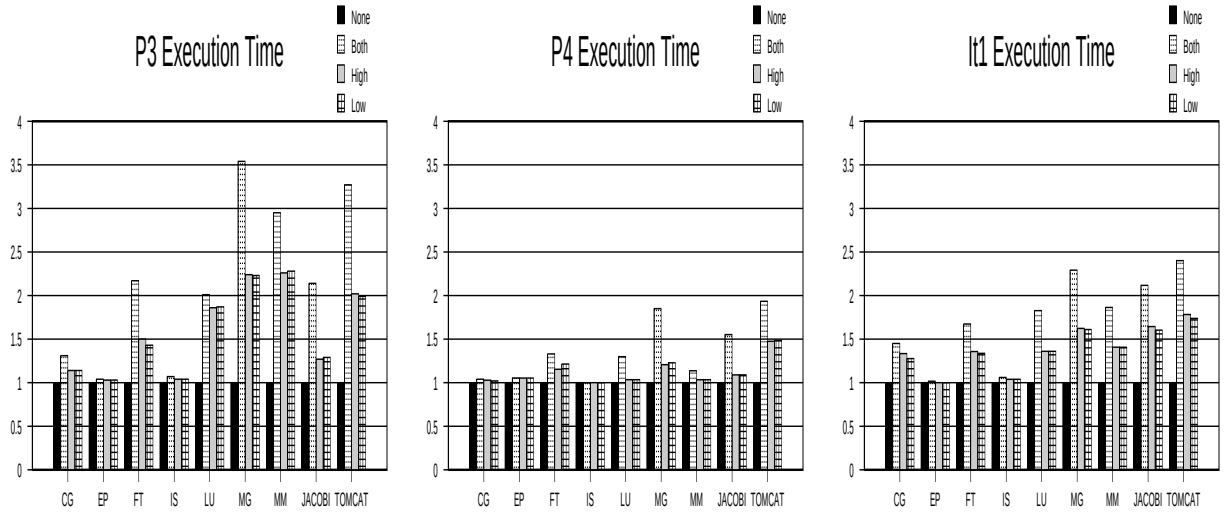

Figure 6: Graph of normalized execution times on the Pentium 3, Pentium 4, and Itanium 1.

4.1 Experimental setup

We performed our experiments on several different architectures: 2.4 GHz Pentium 4 running Redhat Linux 7.3, 2GB main memory, 512KB L1 cache; 1GHz Pentium 3 running Redhat Linux 7.3, 256MB memory, 256KB L1 cache; 733 Mhz Itanium 1 running Debian Linux 2.5, 2GB memory, 16KB L1 cache. For all measurements we took 5 samples, discarded the high and the low values, and computed the arithmetic mean.

In our experiments, we break down the costs of doing each component of the array bound check as well as examine the overall cost of performing all array bounds checks. The *None* bar in each of the graphs represents the original code with no bounds checking performed. All performance measurements are normalized to this value. The *Both* bar in each graph represents the cost of performing both a high and a low bound check for each array access. The *High* bar represents the cost of performing *only* the high bound check, and the *Low* bar represents the cost of performing *only* the low bound check on each array access. The latter two bars represent potential optimizations where *one*, but not both, bounds checks can be (mostly) eliminated. This might be possible if, for example, the array index expression is monotonically decreasing (or increasing) inside of a loop.

4.2 Results

Figure 6 shows the execution time for each benchmark on the three architectures. Overall, on the Pentium 3, the cost is the largest, averaging 117%. On the Itanium, the cost is significant, averaging 74% across all benchmarks. On the Pentium 4, the cost is somewhat less, averaging 34%. We examined on-chip, low-level performance counters to further analyze our benchmarks.

First, the bounds checking versions performed better than we expected when considering the

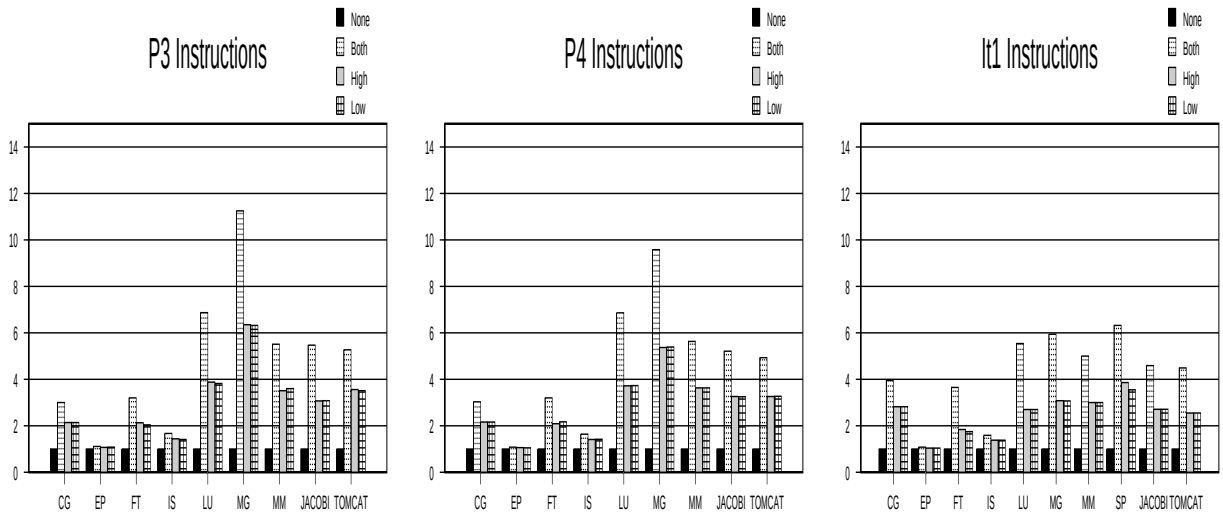

Figure 7: Graph of normalized instructions on the Pentium 3, Pentium 4, and Itanium 1.

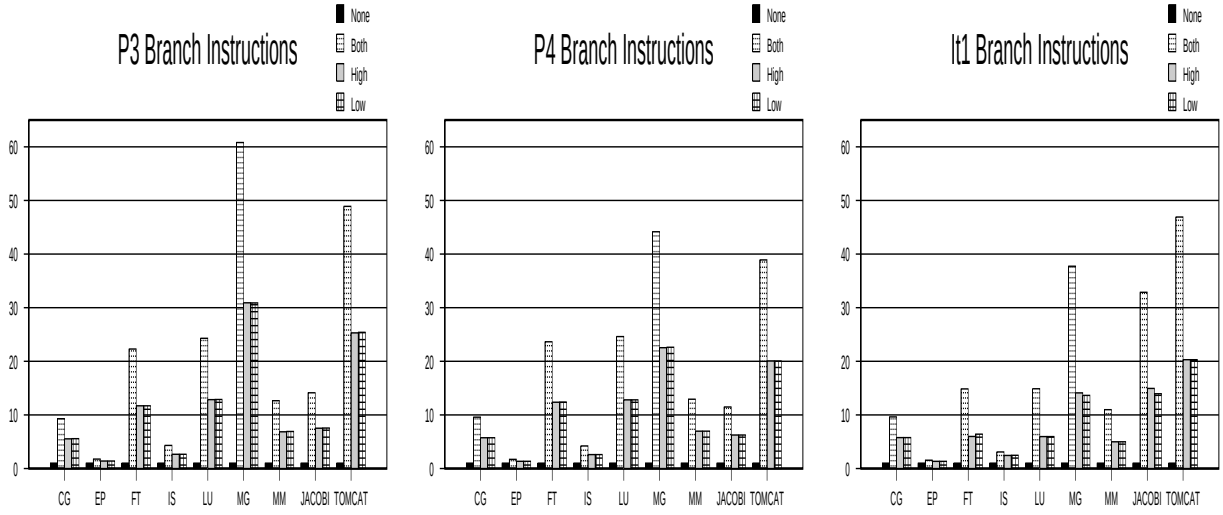

Figure 8: Normalized branch instructions on the Pentium 3, Pentium 4, and Itanium 1.

instruction counts (see Figure 7). In nearly every case, the number of instructions executed more than doubled with bounds checking, in some cases increasing by nearly an order of magnitude. Despite this increase, the execution time does not increase proportionally. We attribute this to two primary factors: branch prediction and low instruction-level parallelism (ILP). Many of the additional instructions are branches required by bounds checks (see Figure 8). In our benchmarks, there were no array access violations, so the hardware could usually predict these branches correctly. In addition, on all three architectures, many of the array bounds checks are overlapped with computation. This is because the effective ILP is low in the original code.

Second, we see that the overhead for a single (either low or high) bounds check is often (but not always) less than half the cost of performing both bounds checks. Examination of the low-level counters reveals the cause of some of this difference. Despite the larger number of instructions in the version with two bounds checks, there are still more pipeline stalls than in the version with one bound check. This makes investigation of optimizations that eliminate one of the two bounds checks potentially useful.

Third, the significant difference between the Pentium 3 and Pentium 4 is surprising; the counters do not reveal a single obvious cause. However, we found that while the assembly code is identical, there are more memory references and branches in the Pentium 3 execution. We do not as of yet have an explanation for this.

Finally, the effect of array bounds checks on the Itanium 1 is particularly pronounced. On VLIW architectures such as the Itanium 1, the compiler is responsible for finding independent instructions and placing them into a 3 instruction bundle. Because effective ILP levels are often low, bundles frequently contain `nops`—no productive code can be inserted into the bundle. Hence, these `nops` can sometimes be replaced with components of array bounds checking code (e.g., loads, comparisons, or branches). However, the compiler could not overlap all bounds checking code, which results in the substantial overhead shown in Figure 6. Another potential benefit provided by the Itanium hardware is predication, including inserting multiple branches into the same bundle (effectively performing two bounds checks at the same time). Inspection of assembly code indicated that `bcc` rarely did this in practice. It should be noted that any independent improvements in compiler technology to improve ILP will decrease the opportunity for inserting bounds checking code into existing bundles.

5 Conclusion and Future Work

This abstract has examined the costs of bounds checking in a set of scientific programs. Our results show that the overhead due to these checks can be significant, especially on machines that do not have significant support for speculative execution. In particular, the Pentium 3 and the Itanium suffered large overheads, averaging 117% and 74%, respectively. Meanwhile, the Pentium 4 had much less overhead, averaging 34%.

This abstract has reported our current project status; our performance results show that techniques to reduce the overhead of array bounds checks may result in a significant performance gain. We are currently investigating, on 64-bit machines, a combination of compiler analysis and OS-supported memory layout to reduce bounds checking overhead to a minimum. Our focus is on

programs that are not amenable to static analysis. As we believe that many users will move to 64-bit machines, these techniques could lead to the acceptance of array bounds checking in the scientific community.

References

[1] Aleph One. Smashing the stack for fun and profit (http://destroy.net/machines/security/p49-14-aleph-one).

[2] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[3] 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.

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

[5] 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.

[6] P. Artigas, M. Gupta, S.P. Midkiff, and J.E. Moreira. Automatic loop transformations and parallelization for java. In *International Conference on Supercomputing*.

[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 Impl.*, pages 182–195.

[10] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, Jan 1998.

[11] Tzi cker Chiueh and Fu-Hau Hsu. RAD: A Compile-Time solution to buffer overflow attacks. In *International Conference on Distributed Computing Systems*, Apr 2001.

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

[13] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.

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