Architectural Support for Safe Software Execution on Embedded Processors [*]

Divya Arora
Dept. of Electrical Engineering
Princeton University,
Princeton, NJ 08544
divya@princeton.edu

Anand Raghunathan,
Srivaths Ravi
NEC Laboratories America
Princeton, NJ 08540
{anand, sravi}@nec-labs.com

Niraj K. Jha
Dept. of Electrical Engineering
Princeton University,
Princeton, NJ 08544
jha@princeton.edu

ABSTRACT

The lack of memory safety in many popular programming languages, including C and C++, has been a cause for great concern in the realm of software reliability, verification, and more recently, system security. Despite their limitations, the flexibility, performance, and ease of use of these languages have made them the choice of most embedded software developers. Researchers have proposed various techniques to enhance programs for memory safety; however, they are all subject to severe performance penalties, making their use impractical in most scenarios. In this paper, we present architectural enhancements to enable efficient, memory-safe execution of software on embedded processors. The key insight behind our approach is to extend embedded processors with hardware that significantly accelerates the execution of the additional computations involved in memory-safe execution. Specifically, we design custom instructions to perform various kinds of memory-safety checks and augment the instruction set of a state-of-the-art extensible processor (Xtensa from Tensilica, Inc.) to implement them. We demonstrate the application of the proposed architectural enhancements using CCured, an existing tool for type-safe retrofitting of C programs. The tool uses a type-inferencing engine that is built around strong type-safety theory and is provably safe. Simulations of memory-safe versions of popular embedded benchmarks on a cycle-accurate simulator modeling a typical embedded system configuration indicate an average performance improvement of 2.3×, and a maximum of 4.6×, when using the proposed architecture. These enhancements entail minimal (less than 10%) hardware overhead to the base processor. Our approach is completely automated, and applicable to any C program, making it a promising and practical approach for addressing the growing security and reliability concerns in embedded software.

Categories and Subject Descriptors: B.8 [**Hardware**]: Performance and Reliability
General Terms: Performance
Keywords: Extensible processors, memory safety, type safety

1. INTRODUCTION

The design of the C programming language in the 1970s sacrificed type and memory safety in the interest of speed, flexibility, and memory usage. This resulted in a language which imparts a fine granularity of control over data representation, memory management, *etc.*, but is intrinsically unsafe and susceptible to memory access errors, if this control is used without caution. However, for developers, the features and ease of use of C/C++ often outweigh their lack of memory safety mechanisms. Hence, they continue to be the languages of choice for a wide range of software. This has motivated research on techniques to address these limitations without requiring too much effort on part of the developer.

A *memory access error* is defined as a read or write of a memory location, through dereference of a pointer or subscripted array, that resides outside the scope of the referent. The scope of a referent is restricted both spatially and temporally – *i.e.*, an access derived from the referent is valid only for a definite range of memory locations and between definite program points during execution. Violation of the spatial scope results in errors such as writing past the end of an array, reading uninitialized memory, null pointer dereference, *etc.* Violation of the temporal scope is manifested in the form of accesses to memory chunks that have been deallocated, either implicitly or explicitly. Other forms of memory errors include memory leaks (losing a pointer to a memory location and hence rendering it inaccessible by the program), misuse of casts, type mismatch between run-time and static type of variables, *etc.*

The abundance of pointers and NULL-terminated strings in C programs facilitates the occurrence of memory errors. Complex pointer arithmetic is frequently the cause of off-by-one errors. Many C library functions (*e.g.*, `strcpy`) rely on the NULL-terminator of strings to determine the number of iterations of loops that traverse them, resulting in an out-of-bounds access if the terminator is not present. The ANSI C standard allows a program to increment a pointer to point to a memory location past the end of an array to test for the end of the allocated buffer. With many out-of-bounds pointers prevalent in programs, it is easy for a programmer to dereference them in error.

Memory-safety violations can be very subtle and difficult to catch. In the best case, they expose themselves by leading to a program crash during testing. The more insidious bugs may be manifested only in rare, hard-to-reproduce cases or may appear along complex program paths that obscure the correlation between the error and its source. However, their impact is not limited to program crashes. The absence of memory safety is a leading cause of security vulnerabilities in C programs. Buffer overflow attacks, that exploit these weaknesses, have emerged as one of the most common causes of security violations. An analysis of the advisories published by CERT [1] over a three month period, Jan.-Mar. 2006, indicates that 23 of the 67 vulnerabilities were pertaining to buffer overflows.

While most studies for enhancing memory safety and security have been performed in the context of general-purpose computers, embedded systems, which are used to perform a wide range of critical functions, represent an equally if not more significant platform. Technological advances in the field of embedded computing systems have increased their capabilities manifold, resulting in their pervasive use. Embedded software content has been growing rapidly in order to keep up with end-user functionality and performance requirements. Increasing complexity of software, together with shorter design cycles, implies that several bugs and vulnerabilities go undetected during the software design process. The addition of

[*]This work was supported by NSF under Grant No. CCR-0326372.

features such as network connectivity and extensibility (ability to extend installed software or download new software) causes security concerns for embedded systems to assume significant dimensions [2].

1.1 Paper Overview and Contributions

In this paper, we present architectural enhancements for efficient, memory-safe execution of programs on embedded processors. We augment Xtensa LX, a state-of-the-art configurable and extensible processor core from Tensilica, Inc. [3], with custom hardware that accelerates a wide range of memory-safety checks. Unlike the traditional use of custom instructions to accelerate application-specific hot spots, the proposed use of custom instructions is broader in scope since the need for memory safety transcends application domains.

We consider the CCured [4] framework, which automatically analyzes and transforms C programs into memory-safe versions. CCured is based on a strong type-inference theory and produces C programs that are provably safe. As an embodiment of the proposed concepts, we have implemented a suite of custom instructions, called `XCHECK`, that efficiently implements the computations involved in the dynamic checks used in CCured. We have developed a tool that takes any program generated by the CCured tool, and modifies it to use the proposed custom instructions for memory-safety checks. We also show that the area overhead of the additional hardware is minimal (an increase of around 10% in the processor core without caches). *To the best of our knowledge, this is the first work that proposes comprehensive hardware support for memory-safe execution of software on embedded processors.*

The remainder of this paper is organized as follows. We present a survey of relevant past work in Section 2. Section 3 describes preliminaries of memory safety checks and Section 4 details the architectural modifications proposed for the same. We present our experimental methodology and results in Section 5 and conclude in Section 6.

2. RELATED WORK

Software-based memory-safety techniques can be broadly classified into two categories - static analysis and dynamic checking. Static analysis tools usually formulate the memory-safety checking problem as a constraint satisfaction problem which is then solved using generic or domain-specific methods. Constraints are generated using data-flow analysis which ranges from simple (context-insensitive, flow-independent) to extremely complex (inter-procedural, context-sensitive, flow-sensitive). Some of the recent static analysis techniques are described in [5–7]. An important advantage of static techniques is that bugs can be caught before the code is deployed, thus eliminating the execution time overhead resulting from run-time checks. However, they do not guarantee completeness, suffer from a large number of false positives and do not scale to larger programs.

Dynamic techniques instrument the program (source code, object code or executable) to perform memory-safety checks at runtime. One of the earliest tools in this category is Purify [8] which uses object code insertion to check for memory leaks and access errors. In [9], the authors introduced an extended pointer representation known as *safe pointer*, to incorporate attributes such as size, storage class, lifetime *etc.*, within the pointer data type. These attributes are used to perform range checking when dereferencing the pointer or performing pointer arithmetic. The advantage of dynamic approaches is that all values (pointer offsets, buffer lengths) and addresses are known at runtime and only feasible program paths are considered. The biggest drawback is the performance penalty entailed by the checks, which reduces most of these tools to debugging aids.

The security risks faced by unsafe C software have inspired many countermeasures aimed at preventing buffer overflows. Since the objective is narrower, many techniques restrict themselves to a subset of buffers (*e.g.,* stack buffers, strings) or known vulnerabilities [10–12], sacrificing code coverage and security for performance.

Recently, a promising direction was demonstrated by CCured [4,13], a hybrid technique that employs static type-inferencing and run-time checking to create a type-safe version of a C program. Static analysis reduces the need for run-time checks in some parts of programs, thereby reducing the associated overheads. Cyclone [14], a dialect of C, with syntax similar to C, also uses similar principles of combining static and dynamic techniques. We base our work on CCured, since it is provably safe and provides an automated mechanism to convert regular C programs to their CCured version. We discovered that although CCured's static analysis reduces the number of inserted checks, resulting in substantial performance improvements over traditional run-time techniques, it still causes a significant performance loss even for medium-sized programs, leaving much room for improvement. These overheads restrict its applicability, and may preclude its use in embedded systems that are performance-critical or contain processors with limited computational capability.

We are not aware of any research on comprehensive hardware support for efficient, memory-safe software execution. However, researchers have proposed architectural features to detect security violations that could result from lack of memory safety. Hardware support for the prevention of stack overflows was proposed in [15]. Hardware-based run-time monitoring techniques to detect security violations were proposed in [16–18]. These techniques do not guarantee memory-safe program execution – rather, they focus on specific side-effects such as control-flow violations. The closest related work to ours is the work of Patil *et al.* [19], who propose the use of a second dedicated processor to perform bounds checks for memory accesses. Our work addresses the overhead of memory safety through a different approach – the addition of a small amount of hardware in the form of custom instructions – which, we believe, is more suited to the domain of resource-constrained embedded systems.

3. BACKGROUND

A program's execution is considered memory-safe if it validates each memory access that is made through a pointer or array, *before* the access occurs, and prevents it if determined to be invalid, possibly terminating the program and outputting the location and source of error. In this section, we briefly explain how memory safety is achieved in the context of the CCured framework.

CCured is built on top of a source-to-source translation framework. The core of CCured is composed of an inferencing engine which analyzes the source code and inferences the *class* of each pointer depending on its use, and the operators applied to it. The class of a pointer determines a set of properties or invariants that must be satisfied before it is dereferenced. CCured uses a combination of static analysis and dynamic checks to ensure that these invariants are satisfied. It inserts calls to checking functions or macros at program locations where the use of a pointer cannot be statically inferred to be safe. The representation of some pointers is also changed to include additional data that are used in the dynamic checks. There are four classes of pointers:

- `SAFE`: These pointers are not subject to any pointer arithmetic or casts in the program. They are represented using the standard C representation of one word, and require minimal checking.
- `SEQ` (SEQuence): These include pointers that are used in pointer arithmetic operations but are not subject to arbitrary casts. Their representation is enhanced to include meta-data (start and end of allocated region), hence they require more storage space than regular pointers. They are also more expensive to use, since each dereference may be accompanied by a bounds-check.
- `FSEQ` (Forward SEQuence): Some `SEQ` pointers can be statically inferred to move only forward during the entire program execution. These are placed under a separate class `FSEQ` which permits them to have lighter-weight checks than `SEQ` pointers.
- `WILD`: These are pointers whose type and size cannot be inferred statically. The representation for these pointers is augmented by storing the start address of the allocated region and designating fields to store run-time type information such as number of words allocated, and

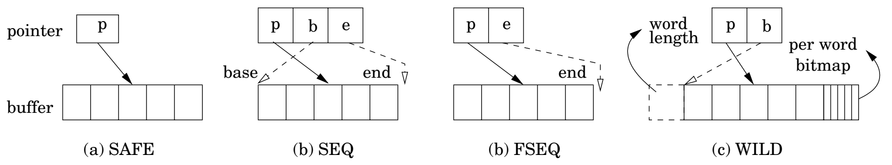


Figure 1: Memory layout of different classes of pointers

a bitmap specifying whether each word in the allocated region is itself a pointer. The compiler inserts code to automatically update this information at run-time.

The memory layout for different classes of pointers is depicted in Fig. 1. The checks that need to be inserted for each pointer dereference/operation depends on the pointer's class. The run-time overhead depends on the number of pointer dereferences that need to be dynamically verified, and the distribution of the pointers across the aforementioned classes.

4. ARCHITECTURAL SUPPORT FOR MEMORY SAFETY

In this section, we provide an overview of our approach for reducing the execution time overheads incurred by memory-safety checking, detail the proposed architectural enhancements, and describe the software flow that is used to automatically enhance programs for efficient, memory-safe execution.

4.1 Overview of Proposed Approach

Fig. 2 presents an overview of the proposed approach. We augment the base processor core with a group of custom instructions, known as XCHECK instructions, to efficiently perform a range of checks involved in memory-safe execution. The shaded box next to the base processor represents the additional hardware. The Xtensa LX processor allows designers to add not only logic to the datapath, but also include additional pipelines, define their own execution units, load/store units, ports, *etc.* The hardware that implements custom instructions is described in Tensilica's proprietary language, TIE [3]. Ideally, XCHECK instructions should be designed to perform the same checking as software checking functions and raise a software exception in case of a violation. However, there is no easy way for custom instructions to access the processor status register in Xtensa. Therefore, we add a new 4-bit custom register, called EXCP (EXCePtion), to the processor, which is set by XCHECK instructions to specified error values. This register is exported through an output port and fed back to the processor as an external interrupt. Thus, the processor receives an interrupt whenever an access violation is detected, and the value of the interrupt indicates the kind of error encountered. The following sections describe the design of XCHECK instructions, and the software flow used to generate memory-safe programs, in greater detail.

4.2 Architectural Details

At the core of the proposed architecture are the single-cycle XCHECK instructions that perform various kinds of memory-safety checks. The custom instructions are designed to exactly emulate the functionality of checks used in CCured, in order to preserve the memory safety guaranteed by its type-inferencing engine. However, the interface to an instruction may differ from that of the corresponding checking routine depending on the implementation. Some software checks have a one-to-one correspondence with a single XCHECK instruction, while others have to be broken down into a sequence of instructions due to the constraints imposed by the processor architecture and compiler.

Table 1 describes the custom instructions used to perform memory safety checks. For each instruction, the table lists the instruction name and usage, its function, and the class of pointers that it applies to. SAFE pointers are subject to three main checks – xcheck_null (ensure that a pointer is not null when it is dereferenced or used to compute the address of a sub-object it points to), xcheck_retptr (verify that the address of a local variable is not returned by a function) and xcheck_storeptr (described below). SEQ pointers are accompanied with bounds checks whenever they are dereferenced or

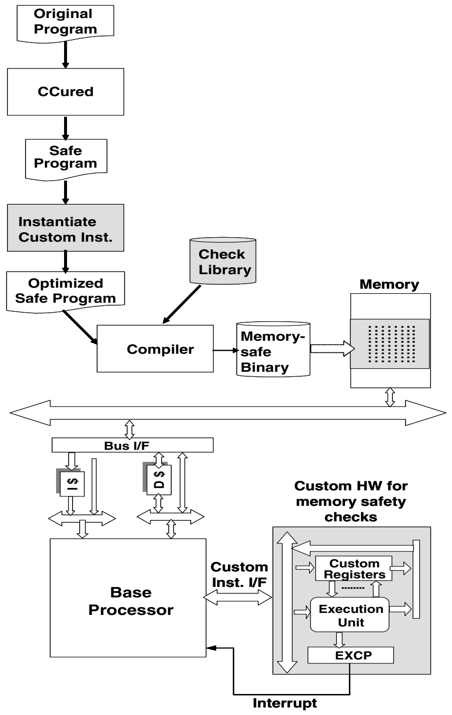


Figure 2: Overview of proposed approach for efficient memory-safe execution

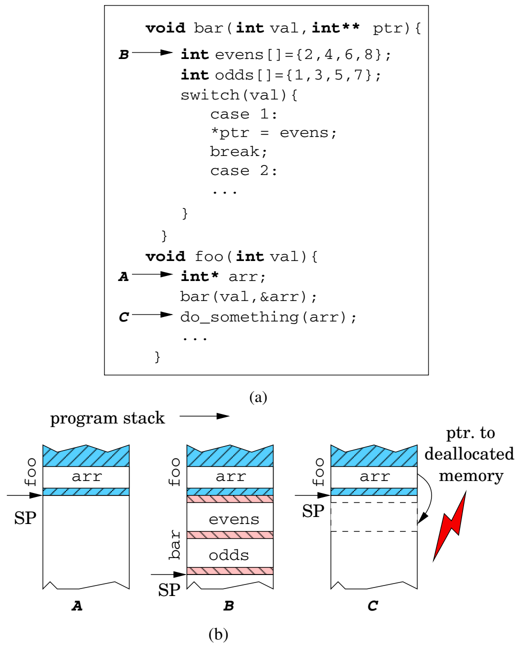
cast to SAFE pointers. The checking instruction takes care of various special cases such as permitting the pointer to be cast to arbitrary integers, allowing access of empty structures, *etc.* FSEQ pointers use similar bounds checks except comparison is performed only against an upper bound. The xcheck_align instruction ensures that the pointer stays aligned with the boundaries of the allocated region after casting.

As mentioned earlier, WILD pointers are accompanied with a tagged area which contains a bit for each word in the allocated region, denoting whether it is a pointer or not. To maintain consistency of this area, WILD pointer tags are updated each time they are read or written to using XCHECK instructions (described in the last three rows of Table 1). Due to space limitations, we do not go into the details of all the above checks and their usage in application programs. Instead, we walk through one instance of unsafe execution and the corresponding XCHECK instruction employed to prevent it, in Figs. 3 and 4.

Example: A common source of unsoundness is when the address of a local variable is returned by a function or stored into a pointer residing in the heap, global data space or a higher stack frame. Fig. 3(a) shows a code snippet of a program, *store.c* with such an error. The stack layout at the marked program points is shown in Fig. 3(b). At points A and B, the stack contains the frames of only foo() and both foo() and bar(), respectively. bar() is passed a parameter val and the address of an integer pointer, ptr. It assigns ptr to point to the address of an appropriate integer array depending on the value stored in val. However, the arrays evens[] and odds[] are declared on the stack frame of bar(), and after it returns, this space is deallocated and becomes an invalid target for a pointer. Now, arr points to garbage values and its use after point C will lead to unexpected program behavior.

Table 1: `XCHECK` custom instructions for memory safety checks

Instruction	Function/comments	Class
`xcheck_null` r_p	Ensure pointer is not null. r_p = pointer.	`SAFE`
`xcheck_retptr` r_p,r_{sp},r_{top}	Ensure that a function does not assign the address of a local variable to a returned pointer. r_p = returned pointer, r_{sp} = stack pointer, r_{top} = top of current stack frame.	`SAFE`
`xcheck_storeptr` r_{where},r_{sp},r_p	Ensure that a function does not assign the address of a local variable to a global or heap location. r_{where} = address of the pointer written to, r_{sp} = stack pointer, r_p = pointer that is copied.	`SAFE`
`xcheck_seq00` r_e,r_p `xcheck_seq01` r_b,r_e,r_p,n `xcheck_seq11` r_b,r_e,r_p,n `xcheck_seq10` r_b,r_e,r_p,n	Check if a SEQ pointer is within bounds. r_p = pointer, r_b,r_e = base, end of allocated region, n = table constant for size. These instructions are mutually exclusive.	`SEQ`
`xcheck_fseq0` r_e,r_p,n `xcheck_fseq1` r_e,r_p,n	Check if an FSEQ pointer is within bounds. r_p = pointer, r_e = end of allocated region, n = table constant for size. These instructions are mutually exclusive.	`FSEQ`
`xcheck_seq2fseq` r_b,r_p	Check when a `SEQ` pointer is cast to `FSEQ` pointer. r_b = base of allocated region, r_p = pointer.	`FSEQ`
`xcheck_align` r_e,r_p,r_{sz}	Check if pointer is aligned with the end of allocated region. r_p = pointer, r_e = end of allocated region, r_{sz} = size of the type accessed.	`(F)SEQ`
`xcheck_fetchlen` r_b,r_p,r_{len}	Fetch size of the area pointed to by a `WILD` pointer into r_{len}. r_p = pointer, r_e = end of allocated region.	`WILD`
`xcheck_wildrd1` r_b,r_p,r_{words} `xcheck_wildrd2` r_p	Check tags when dereferencing a `WILD` pointer. r_p = pointer, r_b = base of allocated region, r_{words} = number of words in allocated region.	`WILD`
`xcheck_wildwr1` r_b,r_p,r_{words} `xcheck_wildwr2` r_{cond} `xcheck_wildwr3; xcheck_wildwr4`	Set tags when writing a `WILD` pointer. r_p = pointer, r_b = base of allocated region, r_{words} = number of words in allocated region. The last two instructions are executed if `xcheck_wildwr2` returns $r_{cond} = 1$.	`WILD`



```
void bar(int val,int** ptr){
B   int evens[]={2,4,6,8};
    int odds[]={1,3,5,7};
    switch(val){
       case 1:
       *ptr = evens;
       break;
       case 2:
       ...
    }
 }
void foo(int val){
A   int* arr;
    bar(val,&arr);
C   do_something(arr);
    ...
 }
```

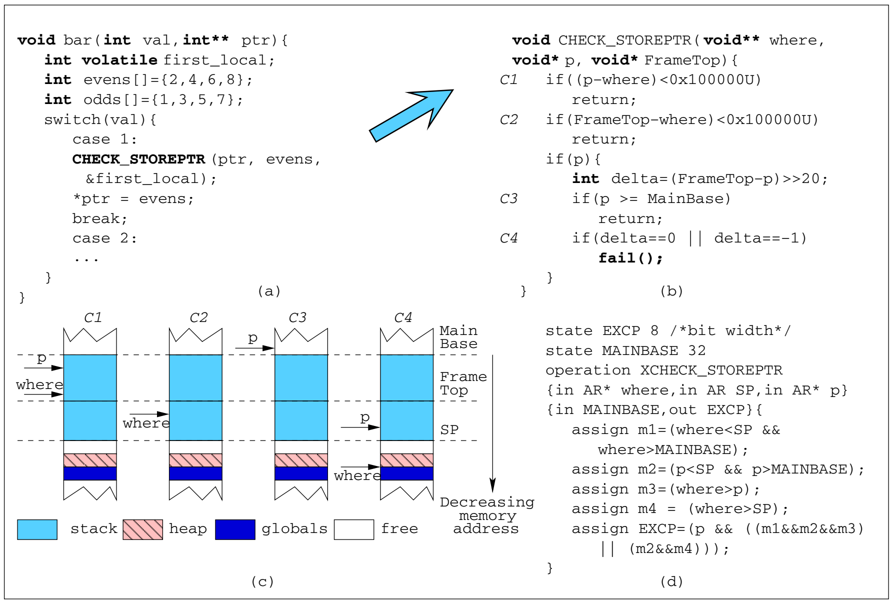
Figure 3: (a) Code snippet from an example program *store.c* performing an invalid pointer write, and (b) stack layout at different program points in the execution of *store.c*

Fig. 4(a) shows the CCured version of *store.c*. A variable declaration, `int volatile first_local` is added at the entry of function `bar()` to mark the top of its stack frame. Additionally, a call to a check function, `CHECK_STOREPTR`, shown in Fig. 4(b) is inserted before `ptr` is written. `CHECK_STOREPTR` uses some compiler-specific approximations to validate an assignment of the form `*where=p`. It comprises four conditionals, each of which is illustrated in Fig. 4(c). *C1* refers to the case when `where` is less than and within a pre-specified distance of `p`; therefore both `where` and `p` are addresses in the program stack, and the store of an address in a pointer lower in the stack is permitted. *C2* is the case when `*where` is in the current stack frame and any store to it is permitted. *C3* is a special case since assigning pointers to addresses above the frame pointer of `main()` (which store command line arguments), as denoted by `MainBase`, is permitted. An error occurs in case *C4*, when `*where` resides in the heap/global space and `p` is in the stack.

Fig. 4(d) shows the TIE code for the custom instruction, `xcheck_storeptr`. The instruction uses three register inputs, one custom register input (`MAINBASE`) and sets the exception register `EXCP` to 1 in case of a violation. It uses the actual stack pointer `SP` for address comparisons and eliminates all approximations made in the software check. The instruction flags an exception if both `p` and `*where` lie on the program stack but `where > p`, or if `p` lies in the stack area but `*where` belongs to the heap/global space. The instruction is more accurate than the original CCured software implementation and speeds up execution by adding hardware to carry out the comparisons in parallel in a single cycle. □

One major limitation of TIE instructions is that they permit a maximum of three input/output operands passed through general-purpose registers. This prohibits implementation of even simple checking routines as a single instruction if they take more than three arguments. In addition to this, some software checking functions are too complex functionally to be implemented as a single instruction. We use several techniques to handle such cases:

- Decomposition: A simple function of the form `fn(x,y, z,b1,b2)` where `b1,b2` are Boolean parameters, whose values are statically known, can be decomposed into four instructions for each of the combinations: `(b1,b2)=(0,0), (0,1),(1,0),(1,1)`. The four instructions are designed for these special cases. Also, they are mutually exclusive and can thus share hardware among themselves.
- Specialization: An extension of the above technique is when a parameter takes a limited number of values (but more than just two). In such cases, we designed instructions that are specialized versions of checking routines, *i.e.*, they assume fixed values for one or more parameters. TIE permits storing a table of constants in hardware, which can then be accessed by instructions, in addition to the three register operands. One example found in our tests were functions such as `fn(x,y,z,size)`, where the argument `size` was usually `sizeof(datatype)`. The sizes of standard data types are fixed and known *a priori*. Whenever the checking routine is called with a



```
void bar(int val,int** ptr){
   int volatile first_local;
   int evens[]={2,4,6,8};
   int odds[]={1,3,5,7};
   switch(val){
      case 1:
      CHECK_STOREPTR(ptr, evens,
       &first_local);
      *ptr = evens;
      break;
      case 2:
      ...
   }
}
```

(a)

```
    void CHECK_STOREPTR(void** where,
    void* p, void* FrameTop){
C1     if((p-where)<0x100000U)
          return;
C2     if(FrameTop-where)<0x100000U)
          return;
       if(p){
          int delta=(FrameTop-p)>>20;
C3        if(p >= MainBase)
             return;
C4        if(delta==0 || delta==-1)
             fail();
       }
   }
```

(b)

(c)

```
state EXCP 8 /*bit width*/
state MAINBASE 32
operation XCHECK_STOREPTR
{in AR* where,in AR SP,in AR* p}
{in MAINBASE,out EXCP}{
   assign m1=(where<SP &&
      where>MAINBASE);
   assign m2=(p<SP && p>MAINBASE);
   assign m3=(where>p);
   assign m4 = (where>SP);
   assign EXCP=(p && ((m1&&m2&&m3)
      || (m2&&m4)));
}
```

(d)

Figure 4: (a) Code snippet from example program *store.c* augmented with CCured checks, (b) checking routine, (c) illustration of various cases handled by the checking routine, and (d) `XCHECK` instruction code in TIE language

constant value for the `size` parameter that matches one of the specialized cases, it is replaced by the corresponding TIE instruction. Otherwise, the check is left as is. In general, our framework permits transparent intermixing of optimized and original versions of checking routines.

- Serialization: Sometimes, functions have to be broken down serially into a sequence of instructions. If the resulting instructions are not completely disjoint in functionality, they have to share parameters, either through explicitly declared program variables or through additional custom registers. TIE instructions cannot be used directly to alter the control flow of a program, *i.e.*, they cannot access the program counter. Therefore, control flow has to be added to the sequence of instructions in an indirect manner. For example, a function that performs a complex computation to generate a value `v1` and a condition `c1` for writing `v1` to a memory location, can be implemented as two serial instructions, `I1` and `I2`, where `I1` computes `v1` and writes it to a custom register and `I2` computes the condition and implements a conditional store based on it.

The translation of software checking functions to custom instructions is done manually. However, this requires only a one-time effort. Xtensa provides a software interface for custom instructions which permits any program to use these checks. Hence, custom instructions and TIE states can be accessed from regular C or assembly code.

4.3 Software Flow

The software flow used to generate memory-safe programs for execution on the proposed architecture is presented in Fig. 2. A regular C program is converted to its memory-safe version using CCured, which is then passed through our transformation tool, which instantiates custom instruction invocation in the program. The resulting optimized memory-safe program is compiled and linked with a custom instruction library to produce the final executable, which is run on the enhanced processor.

CCured also changes the interfaces to many standard C library (*glibc*) functions and provides wrappers to call them. This is because it changes pointer representation for the `FSEQ`, `SEQ` and `WILD` pointers and interfaces of functions that take these as arguments needs to be modified. The wrapper-code along with other changes to header files in the *glibc* library were ported to the Xtensa processor in order to enable CCured versions of programs to compile with the Xtensa compiler and run on it.

5. EXPERIMENTAL RESULTS

In this section, we report the performance of memory-safe programs on the Xtensa processor with and without the proposed architectural enhancements. We also report the area overheads entailed in implementing the proposed custom hardware. We selected applications from the Olden [20], MediaBench [21] and MiBench [22] benchmark suites. The first suite comprises relatively smaller programs and was used for testing CCured in [4]. The latter two were selected because they are representative of typical embedded processing workloads.

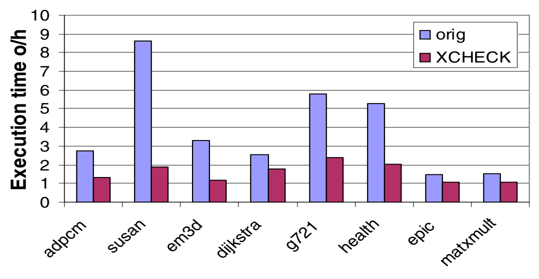
Table 2: Benchmarks used for evaluation and their characteristics

Name	LOC	Checks	Distribution				*
			SAFE	SEQ	FSEQ	WILD	
matxmult	622	23	71	2	27	0	Y
adpcm	642	35	74	0	26	0	
susan	13.1k	1302	74	10	16	0	
em3d	975	66	72	1	26	0	
dijkstra	782	41	91	1	8	0	
g721	3.6k	409	60	2	24	14	
epic	6.2k	798	65	1	34	0	Y
health	2.2k	286	35	1	12	52	

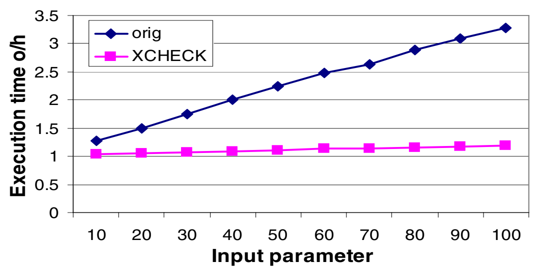
Table 2 lists these benchmarks and some of their characteristics - number of lines of code (LOC) in safety-enhanced code in column 2, number of calls to checking functions in their CCured versions (column 3) and distribution of pointers (columns 4-7). Column 8 is marked “Y” if manual changes were required to the code at any stage. The program *matxmult* required modifications due to an error by the CCured tool, which did not account for changes in pointer representations at one point. *epic* required changes in the original source code as it was found to have an off-by-one error.

5.1 Performance Estimation

All the benchmarks were processed through the software flow described in the previous section and executed on a cycle-accurate simulator for the Xtensa processor. Our base processor configuration comprises a core running at 304 MHz, with 512MB system RAM and 16kB each of instruction and data caches. Fig. 5(a) presents a comparison between the execution time overheads for the benchmarks enhanced with



(a)



(b)

Figure 5: (a) Performance impact of using XCHECK instructions (a) for various safety-enhanced programs, and (b) for *em3d* with varying input size

software memory-safety checking routines, *i.e.*, running on the base processor without the proposed enhancements (marked *orig*), and the same programs running on the base processor with custom instructions for memory-safety checking (marked *XCHECK*). The maximum improvement is obtained for the benchmark *susan*, where the overhead is reduced from 8.63× to 1.87×, an improvement by a factor of 4.6. On an average, performance is improved by about 2.3×. We tested the input independence of our architecture by varying inputs across all benchmarks. The performance comparison for a particular benchmark, *em3d*, with increasing input size is shown in Fig. 5(b).

The above results demonstrate that the proposed architectural enhancements are highly effective in reducing the performance penalty for safe execution in a wide range of embedded software. As a result, we hope that they will enable a broader adoption of memory-safety techniques in embedded systems.

5.2 Area Estimation

The TIE instructions were designed to maximize sharing of hardware and reuse of states between different instructions and thus minimize the area overhead. Calls to checking routines are usually interleaved with regular code. Therefore, hardware sharing does not lead to excessive resource contention (and corresponding pipeline stalls) between XCHECK instructions. The instructions were designed iteratively to permit area reduction without affecting performance. The area estimates were obtained from the TIE compiler.

The area of the proposed architectural enhancements, implemented without any hardware sharing between custom instructions was estimated to be 22,000 gates. The hardware-shared version was estimated to require 11,023 gates, which, when compared to the base processor core (without caches), amounts to an area overhead of around 10%. In the context of a full system-on-chip, the area overhead would be much lower. We believe that this overhead is quite acceptable for most embedded systems.

6. CONCLUSIONS

In this paper, we presented architectural enhancements to an extensible embedded processor to speed up the execution of memory-safe programs. Our enhancements comprised custom instructions for memory access checking that were incorporated into the processor. We based our work on an existing, provably safe, type-inferencing engine. We showed that the proposed enhancements result in notable speedups (up to 4.6X, average of 2.3X) in the execution of memory-safe programs. The hardware overhead of our modifications over the existing processor core is quite minimal. Our framework is completely automated and can be applied to a wide range of embedded software programs.

7. REFERENCES

[1] *Vulnerability Notes Database*. CERT coordination center (http://www.kb.cert.org/vuls/).

[2] P. Kocher, R. Lee, G. McGraw, and A. Raghunathan, "Security as a new dimension in embedded system design," in *Proc. Design Automation Conf.*, June 2004, pp. 753–760.

[3] *Xtensa Application Specific Microprocessor Solutions - Overview Handbook*, Tensilica Inc., 2001. (http://www.tensilica.com)

[4] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy software," *ACM Trans. Programming Languages and Systems*, vol. 27, no. 3, pp. 477–526, May 2005.

[5] Y. Xie, A. Chou, and D. Engler, "ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors," in *Proc. European Software Engineering Conf./Int. Symp. Foundations of Software Engineering*, Sep. 2003, pp. 327–336.

[6] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Software – Practice and Experience*, vol. 30, no. 7, pp. 775–802, 2000.

[7] D. Evans, "Static detection of dynamic memory errors," in *Proc. Conf. Programming Language Design and Implementation*, May 1996, pp. 44–53.

[8] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *Proc. USENIX Conf.*, Jan. 1992, pp. 125–136.

[9] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *Proc. Conf. Programming Language Design and Implementation*, June 1994, pp. 290–301.

[10] C. Cowan *et al.*, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. USENIX Security Symp.*, Jan. 1998, pp. 63–77.

[11] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman, "FormatGuard: Automatic protection from printf format string vulnerabilities," in *Proc. USENIX Security Symp.*, Aug. 2001, pp. 191–199.

[12] N. Dor, M. Rodeh, and M. Sagiv, "CSSV: Towards a realistic tool for statically detecting all buffer overflows in C," in *Proc. Conf. Programming Lang. Design and Implementation*, June 2003, pp. 155–167.

[13] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer, "CCured in the real world," in *Proc. Conf. Programming Language Design and Implementation*, June 2003, pp. 232–244.

[14] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *Proc. USENIX Annual Technical Conf.*, June 2002, pp. 275–288.

[15] J. P. McGregor, D. K. Karig, Z. Shi, and R. B. Lee, "A processor architecture defense against buffer overflow attacks," in *Proc. Int. Conf. Information Technology: Research and Education*, Aug. 2003, pp. 243–250.

[16] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Secure embedded processing through hardware-assisted run-time monitoring," in *Proc. Design, Automation and Test in Europe Conf.*, Mar. 2005, pp. 178–183.

[17] R. G. Ragel, S. Parameswaran, and S. M. Kia, "Micro embedded monitoring for security in application specific instruction-set processors," in *Proc. Int. Conf. Compilers, Architectures and Synthesis for Embedded Systems*, Oct. 2005, pp. 304–314.

[18] T. Zhang, X. Zhuang, S. Pande, and W. Lee, "Anomalous path detection with hardware support," in *Proc. Int. Conf. Compilers, Architectures and Synthesis for Embedded Systems*, Oct. 2005, pp. 43–54.

[19] H. Patil and C. Fischer, "Low-cost, concurrent checking of pointer and array accesses in C programs," *Software – Practice & Experience*, vol. 27, no. 1, pp. 87–110, 1997.

[20] "The olden benchmark suite (v. 1.0)." http://www.cs.princeton.edu/~mcc/olden.html/

[21] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. Int. Symp. Microarchitecture*, Dec. 1997, pp. 330–335.

[22] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. Annual Wkshp. Workload Characterization*, Dec. 2001, pp. 3–14.