Master thesis

SIMD-Swift: Improving Performance of Swift Fault Detection

Oleksii Oleksenko

15. November 2015

Technische Universität Dresden
Department of Computer Science
Systems Engineering Group

Supervisor: Prof. Christof Fetzer
Adviser: MSc. Dmitrii Kuvaiskii

Declaration

Herewith I declare that this submission is my own work and that, to the best of my knowledge, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher education, except where due acknowledgment has been made in the text.

Dresden, 9. November 2015

Oleksii Oleksenko

Abstract

The general tendency in modern hardware is an increase in fault rates, which is caused by the decreased operation voltages and feature sizes. Previously, the issue of hardware faults was mainly approached only in high-availability enterprise servers and in safety-critical applications, such as transport or aerospace domains. These fields generally have very tight requirements, but also higher budgets. However, as fault rates are increasing, fault tolerance solutions are starting to be also required in applications that have much smaller profit margins. This brings to the front the idea of software-implemented hardware fault tolerance, that is, the ability to detect and tolerate hardware faults using software-based techniques in commodity CPUs, which allows to get resilience almost for free. Current solutions, however, are lacking in performance, even though they show quite good fault tolerance results.

This thesis explores the idea of using the Single Instruction Multiple Data (SIMD) technology for executing all program's operations on two copies of the same data. This idea is based on the observation that SIMD is ubiquitous in modern CPUs and is usually an underutilized resource. It allows us to detect bit-flips in hardware by a simple comparison of two copies under the assumption that only one copy is affected by a fault.

We implemented this idea as a source-to-source compiler which performs hardening of a program on the source code level. The evaluation of our several implementations shows that it is beneficial to use it for applications that are dominated by arithmetic or logical operations, but those that have more control-flow or memory operations are actually performing better with the regular instruction replication. For example, we managed to get only 15% performance overhead on Fast Fourier Transformation benchmark, which is dominated by arithmetic instructions, but memory-access-dominated Dijkstra algorithm has shown a high overhead of 200%.

Contents

List of Figures

List of Tables

1 Introduction

Before 2000s, research efforts in the field of dependability (also called “reliability”, “resiliency” and “fault tolerance”) primarily considered applications that can afford extra expenses. Most common of those were systems in which human life or well-being depends on correct operation — so-called “safety critical” systems. One such example is a spacecraft, which usually has incredibly complicated safety requirements and high budgets. For example, Space Shuttle Endeavor has cost approximately $1.7 billion in total [NAS00] and contained five identical general-purpose computers to achieve fault tolerance [Skl76].

Another research direction were enterprise servers. Although they may not be critical for human lives, unavailability of such a server may lead to high expenses. HP NonStop series [McE81] is a vivid example of this domain.

Nowadays, however, things have changed. In the first place, the probability of hardware errors is becoming non-negligible in general. Hardware manufacturers are constantly decreasing feature sizes and lowering operation voltages in order to get higher performance and smaller energy consumption. Such changes allow to get better functional parameters, but they also make hardware much more vulnerable to faults. Microsoft has performed failure analysis on a large testing set [NDO11] and it concluded that machines with at least 30 days of accumulated CPU time have 1 in 190 chance of a crash. After the first failure the probability of a subsequent failure gets up to two orders of magnitude higher. The second observation is that high-end servers are not dominating the market anymore. Instead, components of the shelf (COTS) are becoming a de-facto standard for large-scale systems, such as data centers, because they are much more cost-effective. COTS usually do not have much error protection, except for the basic error correcting codes (ECC) for DRAM. Google has conducted a study of DRAM failures in production clusters [SPW09] and it shows that about a third of machines and over 8% of DIMMs experienced at least one uncorrectable error per year.

The third note is that mission critical systems can be found in commodity products nowadays, such as car control applications and software for autonomous cars. In those fields, budgets are usually much smaller than, for instance, in aerospace, but human lives also depend on execution correctness. For example, if an Electronic Braking System [BM91] fails, a car may crash into an obstacle at full speed.

Historically, the initial approach for dealing with errors was hardware redundancy. The most costly option is to use double or triple modular redundancy (DMR and TMR) with majority voting, but it is also the most reliable one. Therefore it is used, for example, in Boeing 777 [Yeh96]. Cheaper hardware solutions include watchdogs [BS04], redundant hardware threads [Gom+03] or additional logic within a processor [McE81; Sle+99]. Hardware-based solutions are very efficient in terms of performance and normally their overhead is between 5% and 10%. They, however, tend to be too expensive for markets with small profit margins and usually not energy-efficient. In order to overcome these drawbacks, one can avoid using hardware redundancy by applying so-called Software-Implemented Hardware Fault Tolerance (SIHFT), that is, fault tolerance achieved by software-based methods [Gol+06]. Another benefit of this approach is that it can be applied selectively. For example, in modern cars most of the computational power has to be situated under the driver's seat, which is proven to be the safest place in the car [Int14]. It means that a single system executes all computations, both safety-critical, like breaks control, and non-critical, like multimedia. By using SIHFT we can avoid redundancy for non-critical applications and apply it only to critical ones.

Having these benefits, most of SIHFT solutions are based on the idea of duplicated execution, which means that all critical parts of a program are executed twice. It leads to one major disadvantage — even the most optimized state-of-the-art solutions have at least 100% performance overhead caused by duplication. This may be a big issue for some domains. For example, autonomous cars need to process approximately 1 GB of data each second, which leads to very high requirements for computational effectiveness [Int14]. Our goal is to overcome this drawback and to develop a software-based solution with less than 80% overhead.

In this thesis, we introduce SIMD-Swift, a software-based single-threaded approach to achieving data- and partial control-flow fault tolerance. It is based on Error Detection by Duplicated Instructions (EDDI) [OSM02] and its successor Software-Implemented fault tolerance (SWIFT) [Rei+05], which duplicate all instructions and insert periodic checks. Our hypothesis is that employing Single Instruction Multiple Data (SIMD) technology for fault tolerance will improve performance of SIHFT by reducing the number of executed instructions and used registers (decreased register pressure). The main idea behind our approach is to trade time redundancy (duplicated execution) for space redundancy (bigger SIMD registers are used instead of regular registers). SIMD can be considered a commodity hardware since virtually all modern CPUs have it: x86 (Intel, AMD) has SSE and AVX, PowerPC (IBM) has AltiVec, and ARM has Neon.

We implemented SIMD-Swift as a source-to-source compiler and our evaluation shows promising results with benchmarks that are dominated by arithmetic or logic operations. For these types of benchmarks we achieved a performance overhead of at most 75%, and for those that purely consist of arithmetic operations, we managed to get it as low as 15%. Applications that contain mostly control-flow operations or memory interactions show, however, much higher overheads and in general perform worse than SWIFT-hardened.

The thesis is structured as follows. In Chapter 2, we start with a discussion of the existing approaches for handling hardware faults and come to the conclusion that SIMD may be a promising way to improve over the existing solutions. Then we continue with a detailed review of SIMD (in particular, SSE) and ways to use it. In Chapter 3 we describe a general architecture of our approach and explain our assumptions. Chapter 4 goes into details of the implementation and highlights its performance bottlenecks. We also propose an alternative implementation of SIMD-Swift, which offers a trade-off between performance and fault detection capabilities. In Chapter 5, we evaluate the implementation and discuss the drawbacks of SIMD-Swift from both performance and resilience points of view. Chapter 6 considers applicability of the approach and discusses directions of the future work.

2 Background and Related Work

2.1 Related Work

Two broad types of fault tolerance techniques exist: hardware-based and software-based (SIHFT). All of them implement some kind of redundancy, but they do it on different levels: hardware-based approaches employ redundant hardware blocks and most software-based ones use redundant execution.

2.1.1 Hardware-based solutions

Hardware redundancy is a very old technique and it was used in mechanical devices even before first computers. The most basic approach for hardware redundancy is n Modular Redundancy (nMR), including Dual Modular Redundancy (DMR) and Triple Modular Redundancy (TMR). nMR means that all parts of a system have redundant replicas that can be used in case the main one would fail. Although it is expensive, this approach provides a very high level of reliability, which is why it is widely used in aerospace industry. One of the most popular Boeing airplanes, Boeing 777, applies TMR for all main parts of its control system: Primary Flight Computer, communication paths and even electrical power supply [Yeh96].

This approach is also applicable for high-availability servers. HP NonStop Advanced Architecture [Ber+05] consists of two or three SMP Itanium2 server processors working in loose synchronization. Error detection is performed by comparing execution results of the processors. A more conservative way is to add redundancy only to critical blocks in hardware. This approach is widely used among highly-available servers. As one example, IBM S/390 G5 [Sle+99] fully duplicates only the main blocks of the processor — units that handle instruction fetching, decoding, and execution. Unfortunately, all these approaches increase the buying and maintenance cost up to 5 times over commodity hardware.

Instead of replication some approaches use a small and simple coprocessor (watchdog processor) which monitors the main CPU and performs concurrent error detection [MM88]. A similar direction takes the Dynamic Implementation Verification Architecture (DIVA) technique [Aus99] by adding a simple checker module. It verifies the correctness of computation on the core processor and permits only correct results to be written to a storage. Inherent Time Redundancy (ITR) [RR07] also uses a checker, but only for decode and fetch units. Such checker exploits small traces of identical instructions and observes re-occurrence of events that depend purely on instructions. Argus [MBS07] performs dynamic verification of core invariants by a series of hardware checkers. These checker coprocessors are still an active research field, but they did not yet make it to industry.

Parity and Error Correcting Codes (ECC) protection is also a very popular way of achieving reliability, even though it provides only partial protection from faults [SPW09; NDO11]. Parity protection was widely used for register files in platforms such as Intel Itanium [Fet+06] and SunUltraSPARC [KAO05]. ECC modules are used in many high-end RAM chips to check incoming and outgoing data from memory. Parity and ECC, however, only protect storage and not computation which we target in this thesis.

2.1.2 Redundant Multithreading solutions

In between hardware and software solutions lay redundant multicore and multithreaded systems, which make use of readily available multiple execution blocks in modern processors. One of the early efforts in this direction was made in AR-SMT [Rot99], which presents somehow similar approach to DMR, but instead of physical hardware duplication it uses OS-level resources. Main computation is replicated into two threads — one leading and one trailing. The trailing thread repeats the computation and compares produced result with the leading thread.

The following studies tried to improve this approach. Mukherjee et al. [MKR02] reduced the performance overhead of AR-SMT by implementing it in a dual-processor device. Smolens et al. [Smo+04] proposed a set of changes to a conventional superscalar microarchitecture in order to make communication in concurrent error detection more efficient. Wells et al. [WCS09] worked on a problem of mixed-mode computation, that is, a mode in which some applications have high reliability using multithreaded duplication, while other applications run normally without any performance penalty. This technology allows to make a performance-reliability trade-off not on the hardware, but on the application level.

Recent work of Zhang et al. called RAFT [Zha+12] proposes a low-overhead solution based on running a program binary twice and monitoring both instances' behavior at the system call level. Internally, before executing a system call, a first instance compares its arguments with the arguments of a second one. If the arguments match, the system call is executed, otherwise RAFT reports an error and stops program execution. Such approach shows only 2.8% average overhead, which is one of the best performances among multithreaded solutions.

The same idea was implemented at the level of processes as Process Level Redundancy (PLR) [Shy+09]. It works as a software application and does not require any changes neither to hardware nor to OS or target application itself. PLR also provides an additional protection for the memory since memory is duplicated on process replicas.

2.1.3 Software-based solutions

Even though hardware-based approaches are very efficient in terms of performance and expose only 5–10% overhead, they all require specialized or additional hardware, which adds up to the total cost of the system. A more attractive way is to use software-based solutions which essentially come free of cost.

Software-based approaches modify the original program into a functionally similar resilient version with some kind of redundancy that allows to detect errors. It should be noted however, that they all consider only hardware fault tolerance: they assume correct code without bugs such that the faulty behavior can be caused *only* by hardware faults. The main idea behind most of the recent single-threaded software-based solutions is duplicated execution (instruction duplication), that is, all instructions are executed twice and their results are compared. Holm and Banerjee [HB92] were the first ones to investigate this direction, although they kept it mainly theoretical. Their idea was implemented and further developed in EDDI [OSM02]. It benefits from Instruction Level Parallelism (ILP)[1] of modern processors and duplicates not only instructions but also memory state. SWIFT [Rei+05] added control flow checks and removed redundant memory duplication due to the assumption that memory is protected via ECC. ESoftCheck [YGS09] implemented a set of optimizations for SWIFT in order to improve performance. These three works constitute a basis for our thesis, that is why we will take a closer look at them. We will also discuss a bit different Shoestring [Fen+10] approach, which combines instruction duplication with symptom-based error detection.

2.1.3.1 Error Detection by Duplicated Instructions

The main idea behind Error Detection by Duplicated Instructions (EDDI) [OSM02] is to exploit unused ILP resources available in VLIW[2] architectures to run duplicated instructions concurrently. These instructions do not change functionality of the program, but an error in one duplicate most probably will lead to a different result of computation, which can be detected by comparing two results. EDDI also protects data by using different registers and memory addresses for new instructions.

Checking (comparing results of two duplicates) every instruction is an overkill in most cases. First, it will lead to an extremely high performance overhead and second, most of variables are dependent on each other, that is, an error in one variable will propagate to others. For that reason, EDDI performs checks only at points of the program that may influence its output. Those points are:

- program input and output
- memory accesses
- deciding a direction of branch or jump

Consider a simple program performing addition and multiplication, and then printing the result (see Listing 2.1). Listing 2.2 shows a hardened version of the same code. In the first place, all input variables have to be duplicated (lines 2–4). Then, all instructions are executed twice on different replicas (lines 6–9) and result is checked for equality before the output (lines 11–12). In case results are different the program is crashed, since it indicates an error.

[1] Instruction Level Parallelism — an ability to perform multiple operations simultaneously on a single core.

[2] Very Long Instruction Word — a family of processor architectures allowing programs to explicitly specify which instructions will be executed in parallel.

```
// b, c and d are input arguments




a = b + c;

r = d * a;





printf("Result: %d", r);
```

Listing 2.1: Example of EDDI — original code

```
// b, c and d are input arguments
b1 = b; b2 = b;
c1 = c; c2 = c;
d1 = d; d2 = d;

a1 = b1 + c1;
a2 = b2 + c2;
r1 = d1 * a1;
r2 = d2 * a2;

if (r1 != r2)
    crash();

printf("Result: %d", r1);
```

Listing 2.2: Example of EDDI — hardened code

EDDI works with bit-flip faults, that is, it can detect all faults that can be modeled as bit-flips. This includes state changes in memory cells and registers, data corruption in data and address buses, transient errors in functional units and control logic, etc. It also detects control-flow errors caused by branch instruction faults.

EDDI assumes VLIW architecture that allows a compiler to explicitly control parallel execution of instructions. It usually means that a lot of parallelization resources are left unused and can be adopted for executing duplicates. That is the reason why EDDI shows only 80% overhead instead of more than 100% that can be expected from at least twice as much instructions.

Duplication is performed on the assembly source code level, but it is not the only option. Rebaudengo et al. [Reb+01] took another direction and implemented a source-to-source compiler which adds redundancy at a higher level of abstraction — C source code. This adds portability to the solution, but both overhead and fault coverage got worse. Also, this approach works only with all compiler optimizations turned off.

2.1.3.2 Software Implemented Fault Tolerance

The next step in the development of duplicated execution was Software Implemented Fault Tolerance (SWIFT) [Rei+05]. Its main contribution consists of two parts.

First, it adds control-flow protection to EDDI. All code is split into blocks with only one entry point and one exit point — basic blocks. Each of them gets a signature, which is used to detect control-flow faults. One of the general purpose registers keeps a current signature and is used for checking. Every time a program enters another basic block, this register is XOR'ed with a statically determined constant and it gets a value of the current block's signature. This signature is also statically assigned to the block and by comparing with it we can detect control transfer errors. Such protection also makes EDDI's branch validation unnecessary, which improves its performance.

Second, SWIFT assumes ECC-protected memory and caches. It allows to eliminate memory duplication and significantly reduces memory requirements. Moreover, it reduces cache pressure hence increasing performance. All registers, however, are kept duplicated and all values still have to be loaded twice.

This set of optimizations reduced overhead on average to 40% on tests performed by the authors. Such promising results in both SWIFT and EDDI, however, were achieved because of the availability of free ILP resources, which appear if VLIW architecture is used. This architecture was considered promising at that time with Intel introducing its VLIW-based Itanium CPU and many thought it would become standard. Yet, VLIW did not become popular and paper's assumptions do not hold on a much more common nowadays x86 architecture. It means that results from the original papers can not be applied to modern commodity CPUs and we need a better estimation. One of the works that tried to do it belongs to Yu et al. [YGS09] and it shows an average 116% performance overhead for the original SWIFT.

2.1.3.3 ESoftCheck

EDDI and SWIFT insert a check before *every* load, store and branching operation, such that checks account for approximately 40% of the overhead. ESoftCheck [YGS09] applies a set of optimizations in order to remove this redundancy:

- **Recurrent checks.** If a check of one value is always followed by another check of the same value and this value is never changed between them, first one can be considered redundant. It can be removed since an error will always (or at least with very high probability) be detected by the second check.
- **Dependent variables.** Two variables can be called dependent if a value of a second variable is a function of the first one. That means there is no point in checking both dependent variables, since an error from the first variable will propagate to the second and this first check can be removed.
- **Loop checks.** Induction variables and loop invariants can be considered dependent on themselves and any error will propagate to the end of the loop. Thus checking this variables is redundant if we add covering checks at the loop exit.
- **Protected registers.** On some platforms registers are either already hardware-protected or can be protected at low cost. Such protection can detect errors in registers themselves, but not in values that are stored to them as a result of faulty computation. That means, we can remove redundant register checks, but we have to keep checks before stores to prevent faults from propagating to memory.

After these optimizations only checks that are required to detect an error are left. ESoftCheck guarantees that an optimized code will have the same level of reliability as the original code, thus reducing performance overheads at basically no cost.

If we consider x86 architecture without register protection, performance overhead of this approach is on average 102%, which is still not a big improvement. Moreover, ESoftCheck can be partially applied to our SIMD-Swift in order to reduce overhead even further.

2.1.3.4 Shoestring

Shoestring [Fen+10] stays a bit aside from duplicated execution approaches, but it is still worth mentioning since it shows comparably low performance overhead of 16% with a reasonable failure rate of 1.6%. Such results are achieved by combining instruction duplication with a symptom-based error detection. The symptoms can include memory access exceptions, mispredicted branches and cache misses.

This solution is based on the idea that most of transient faults either will not produce an error or will lead to a user-visible failure that can be covered with low-overhead symptom-based detection. For example, a fault in memory will most probably lead to a segmentation fault which is already detected by OS. All the other faults can be covered with instruction duplication using compiler analysis to identify vulnerable parts of code.

The use case of this approach is shifted from high-availability and mission-critical domains towards modern commodity electronics, which starts to experience significant amounts of faults, but does not have high reliability requirements. That is why it provides only opportunistic fault coverage and cannot be used in high-reliability applications.

Table 2.1 gives a comparison of the presented software-based approaches. Columns "Replication Coverage" represent an extent to which a given aspect is covered by an approach. In "Overhead" columns *low* means that performance overhead is less than 50%, *moderate* indicates a range between 50% and 100 % and *high* — more than 100%

Name	**Replication Coverage**					**Overhead**	
	Instruction	Register	Memory access	Memory state	Control transfer	VLIW	x86
EDDI	full	full	full	full	none	moderate	high
SWIFT	full	full	full	none	partial	low	high
ESoftCheck	full	none	full	none	partial	low	high
Shoestring	partial	partial	partial	partial	none	no data	low

Table 2.1: Comparison of main software-based redundancy approaches.

2.2 Background

Our solution relies heavily on the Single Instruction Multiple Data (SIMD) technology. The main idea behind it is to perform the same operation on multiple pieces of data simultaneously (data level parallelism). Figure 2.1 illustrates the difference between traditional and SIMD processing. In general, SIMD adds new, wider registers that are capable of storing several items and the corresponding new instructions that operate on these resisters, computing in parallel on all items.

The first modern implementation of SIMD in processors, called Visual Instruction Set (VIS) [Koh+95], was presented by Sun Microsystems in 1995 as an extension for the SPARC architecture. It was initially marketed as a substitution for discrete video cards and, although it never became one, it has found its niche as an efficient way to optimize

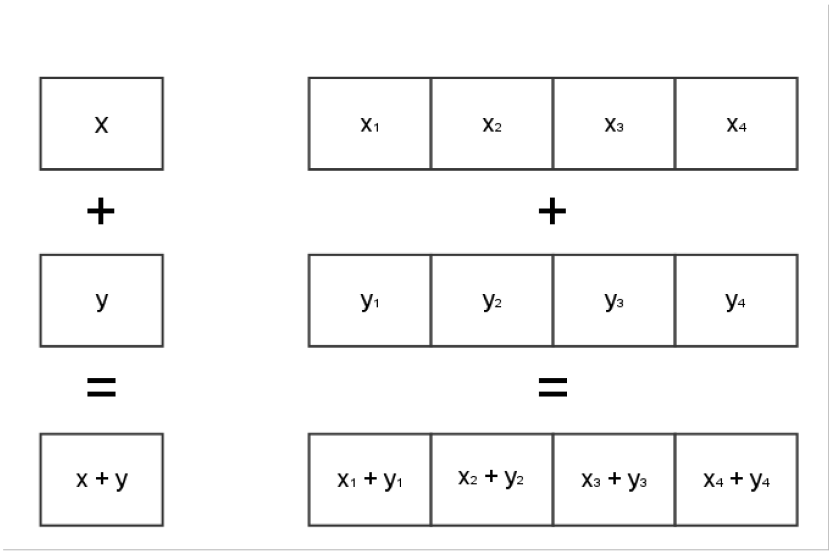

Figure 2.1: Scalar and SIMD processing compared

data processing. Next year, a multimedia extension was also presented for the MIPS architecture [Gwe96].

Yet, it was not before the Intel MMX [PW96] extension for the x86 architecture that SIMD became widely available and used in commodity processors. MMX introduced three packed data types and a set of instructions to operate on them. Data types include packed byte (8 bytes in one 64-bit quantity), packed word (4 words in one 64-bit quantity) and packed doubleword (2 doublewords in one 64-bit quantity). They all use 64-bit registers shared with regular floating-point instructions. Although MMX instructions work with MMX registers, General Purpose Registers (GPR) must be used to specify a memory address operand.

Nowadays, the successor of MMX called Streaming SIMD Extension (SSE) is the most common implementation of SIMD. According to the Steam statistics [Ste15], it is available in 99% of user machines (this number, however, should be treated with care, while gamers tend to have more powerful hardware than average users). Initially SSE was targeted on multimedia applications, in particular on visual and graphical computing. Nevertheless, nowadays it is widely used in all applications that require similar operations on big amounts of data, including digital audio processing, computer vision and even Bitcoin mining.

SSE originally appeared in the Pentium III processor [RPK00]. It gained a lot of popularity by providing compelling performance improvement at a very low cost, since it is much easier and cheaper for processor manufacturers to add another execution block and an extra set of registers than to implement a full-scale core. Still, it is hardly comparable with multicore parallelism — performance boost can be achieved only for certain types of applications and sometimes even only for certain regions of code. That means, SIMD is hardly used in all other applications and can be considered a free resource [Ram12].

From the usage point of view, SSE added a completely separate logical register set to the one available in MMX, which allows to use these technologies concurrently (although, MMX is irrelevant in the scope of this thesis). This set consists of eight 128-bit registers, each able to hold 4 single-precision floating point numbers.

The second generation, SSE2, was intended to fully replace MMX. In order to do that, it introduced 5 new data types: packed double-precision floating point and packed 8-, 16-, 32- and 64-bit integers. SSE2 also contained a set of corresponding instructions to operate on this data.

Afterwards, SSE was evolving more gradually. The third generation introduced a notion of horizontal operations (e.g., horizontal addition), which operate on pairs of values in one SSE register. SSSE3 added a couple of new arithmetic and shuffle instructions. The latest version SSE4, among other new instructions, presented the *PTEST* — an instruction that sets the *Z flag* in status registers (EFLAGS) by performing an AND between its operands, thus allowing to do jumps using packed values directly.

SSE is available only in Intel and AMD processors, but similar technologies can be found in the majority of modern architectures, including the embedded domain. It is implemented in PowerPC as AltiVec [Gwe98] extension, in ARM as Neon [ARM10], and even Atmel microcontrollers have it inside of a Digital Signal Processing (DSP) modules. Since our solution uses a basic idea of SIMD, it can be ported to any of the mentioned architectures with minimal changes. In this thesis, however, we will concentrate on SSE, as the most common technology in general-purpose processors. Specifically, we will use its latest version, SSE 4.2, as it gives the widest functionality and allows us to get the most effective solutions.

Before we go deeper into the details, consider a small example of a common SSE application. Listing 2.3 shows a simple loop written in C without any extensions. This loop performs a single-precision floating-point triad operation on the arrays of size *SIZE*.

```
float a[SIZE], b[SIZE], c[SIZE];
float q;

void triad() {
    for (int i = 0; i < SIZE; i++) {
        a[i] = b[i] + q * c[i];
    }
}
```

Listing 2.3: Original loop

This program can be rewritten using SSE (see Listing 2.4). It will allow to perform 4 triad operations simultaneously in one loop iteration, thus significantly improving the performance. The first change here is the 16-byte alignment of all arrays on the line 3. Such alignment is required because otherwise CPU would have to do extra operations while accessing the data, which would harm the performance. Inside the function on the lines 7–8 we use a new data type, `__m128` — 4 single-precision floating-point values stored in one variable. On the line 8 we use an intrinsic (see Chapter 2.2.2) for the first time, `_mm_set_ps` — replicate the argument (in the given case it is variable `q`) 4 times to make a __m128. Next, as we are working on 4 pieces of data simultaneously, the loop counter is incremented by 4 instead of 1. Before we perform the triad operation itself, we get array elements from the memory on the lines 12–13. The multiplication and the addition operations are replaced by their SSE versions — _mm_mul_ps and _mm_add_ps correspondingly (lines 16–17). On the line 20 result is stored back to the array a.

```
1  #define VECTOR_SIZE 4
2
3  __declspec(align(16)) float a[SIZE], b[SIZE], c[SIZE];
4  float q;
5
6  void triad() {
7      __m128 product, sum, current_c, current_b;
8      __m128 q_packed = _mm_set_ps(q);
9
10     for (int i = 0; i < SIZE; i += VECTOR_SIZE) {
11         // get current elements of arrays b and c
12         current_b = *((__m128 *) &b[i]);
13         current_c = *((__m128 *) &c[i]);
14
15         // calculate triad operation
16         product = _mm_mul_ps(current_c, q_packed);
17         sum     = _mm_add_ps(current_b, product);
18
19         // store result
20         *(__m128 *) & a[i] = sum;
21     }
22 }
```
Listing 2.4: Vectorized loop

In order to better understand the design choices made in this thesis, we will take a more detailed look at the internals of the SSE hardware implementation and at the available instructions.

2.2.1 Hardware implementation

The x86–64 architecture provides 16 128-bit wide registers that are available for the SSE instructions. Figure 2.2 compares them with normal GPRs. It should be noted however, that even though only 16 registers are visible at the assembly level, much more registers (e.g. 168 SSE-AVX registers in Intel Haswell) are implemented physically and can be used for renaming. Specifically, starting from the Intel NetBurst microarchitecture, SSE values are stored in a separate register file that is shared between SSE and floating-point operations.

In the Intel's implementation, SSE instructions are executed by the Floating Point Units (FPU). In the initial implementations (e.g. in the Intel Pentium III) the instruction decoder transformed all 128-bit instructions into pairs of 64-bit microinstructions, which were executed in parallel using ILP available in the super-scalar processors. In the modern implementations (e.g. in the Intel Core 2) SSE has 128-bit dedicated FPUs. In addition to that, processors usually have separate modules for the most frequent or the

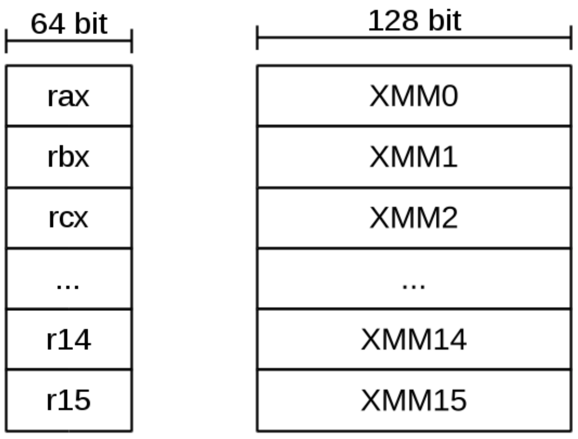

Figure 2.2: GPR and SSE registers compared

most costly operations, such as multiplication, shuffle or memory access. It provides a higher level of parallelism and allows to avoid some common bottlenecks. As one example, an execution unit of the Intel Core 2 microarchitecture (presented in 2006) is shown in Figure 2.3 with SSE parts highlighted.

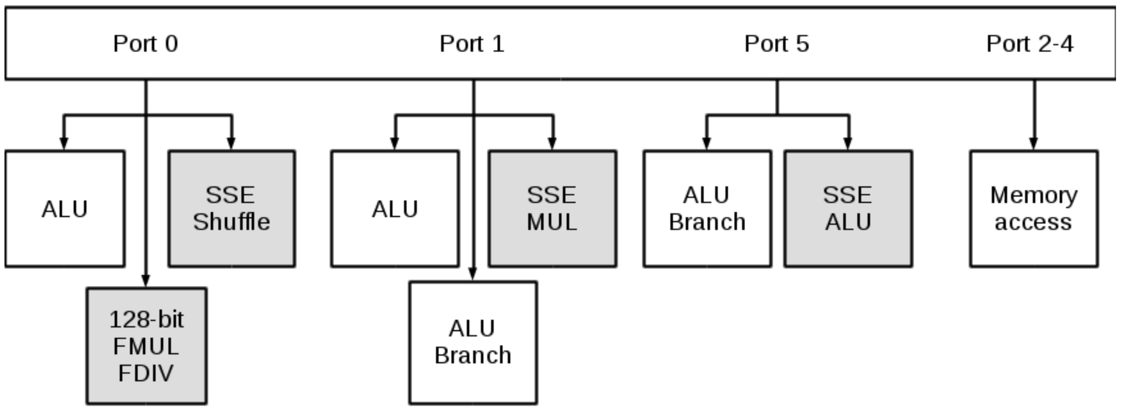

Figure 2.3: Intel Core 2 execution unit
Here: *ALU* — scalar arithmetic logic unit, *FMUL/FDIV* — SSE floating point multiplication and division unit, *SSE MUL* — SSE integer multiplication, and *SSE ALU* — SSE integer ALU.

2.2.2 SSE instruction set and intrinsics

In general, SSE has two modes of operation — it can operate in parallel on all data operands (packed mode) or on the least significant pairs of operands (scalar mode). In other words, in the scalar mode SSE registers operate as normal GPRs. In this thesis, we use the packed mode because it allows to operate on two replicas simultaneously. SSE also provides two modes of floating point arithmetic — IEEE-compliant, which has higher precision and is more portable, and flush-to-zero mode, which has higher performance. In our work, we always assume a default IEEE-compliant mode.

The SSE instruction set can be boiled down to the following instruction types: arithmetic, logic and comparisons, data movement and reorganization, type conversion, state save and restore, memory streaming and caching, media and other special purpose instructions. In this thesis, we use only some instruction types, which we describe in more details in the following.

Arithmetic and Logic. SSE covers most of the arithmetic operations, except the modulo operation. Moreover, multiplication and addition have horizontal versions, that is, they can operate on pairs of numbers inside one register, as shown in Figure 2.4. Logical operations are only implemented in bitwise versions.

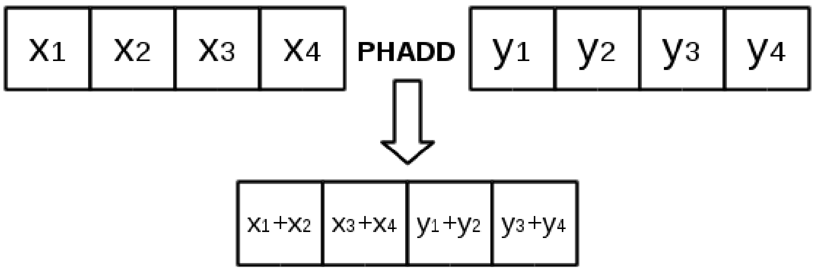

Figure 2.4: Horizontal addition

Comparison. Comparisons act a bit different than their counterparts in the general instruction set. In the first place, they do not set flags in the *EFLAGS* register (used to drive control-flow in x86), which makes an implementation of jumps much more complicated. The only exception is *PTEST*, which sets the *Z flag* in the status register by performing an AND between its operands, thus allowing to do jumps using packed values directly. The second issue is the return value. Instead of returning a boolean, as normal comparisons do, they return either all "1" (if result is "True") or all "0" (in case of "False"). It makes sense in terms of implementation since the comparison is performed on multiple pieces of data and a single boolean value won't be enough to represent a result (like in the case when $x_1 < y_1$ but $x_2 > y_2$). But from the other side, there are no control flow instructions that could operate on these sequences of "1" and "0", which leads to extra efforts for extracting data and consequently to lower performance.

Data movement. SSE can efficiently access a 128-bit wide data in the memory, but the memory address has to be specified using a GPR. It may lead in some cases to additional type conversions and in particular, to an extraction from an XMM register. This is an expensive operation, with a latency of 6 cycles in Intel Haswell.

Data shuffle. Shuffle is a peculiar SIMD operation that performs data rearrangement inside registers. One example of the shuffle is shown in Figure 2.5. In combination with other operations (in particular with horizontal ones), it allows to get much of the functionality that is not implemented in hardware. For example, we can get a horizontal test for equality using a combination of the shuffle, horizontal subtraction and PTEST (see Chapter 4 for more details).

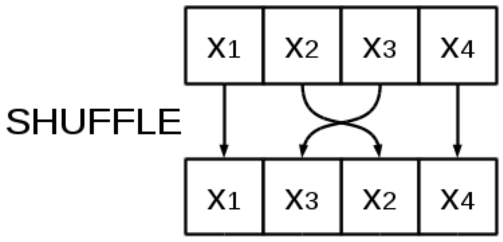

Figure 2.5: Shuffle operation

Type casting and conversion. There are two ways to change a type of a variable in SSE — convert and cast. *Convert* allows to change bit representation of a data, e.g. from integer "123" to floating point "1.23×10^2". *Cast* does not change the bits in data but merely tells a compiler to treat a given value differently. A type conversion is implemented in hardware and quite expensive from the performance point of view, especially for scalar types. Casts, on the other hand, do not take any time at all, since they do not generate any assembly instructions.

Our approach works as a source-to-source compiler and needs an interface to operate with assembly instructions. One option is inlined assembly available, for example, in GCC as a special *asm* keyword. It requires, however, a significant amount of work to reimplement all the functionality available at the assembly level in C code. A much better and easier solution is to use *SSE intrinsics*, which provide a C-level, macro-like interface to abstract the SSE instructions via built-in functions implemented directly by the compiler. Such approach allows to get all the power of SSE without a need to worry about register allocation or code scheduling because the compiler makes all these optimizations by itself.

3 Design

In the previous chapter, we discussed cheap software-based fault tolerance approaches. They provide two advantages: the ability to run on commodity hardware and moderate performance overheads. For example, SWIFT [Rei+05] showed an impressive result of only 40% overhead on VLIW architecture. However, when deployed on a regular x86 machine, its overhead is much worse — 120% as reported by ESoftCheck [YGS09]. The goal of our thesis is to lower this performance overhead and make a single-treaded software-based solution with less than 80% performance overhead on the x86 architecture.

Our approach is to employ SIMD technology (see Chapter 2.2.2) for executing operations on duplicated data, instead of having all instructions executed twice. Such trade-off is valid while SIMD is not used in most applications, excluding media and scientific ones. It also satisfies the availability requirement since SIMD can be considered a commodity hardware — virtually all modern CPUs have it: PowerPC (IBM) has AltiVec, ARM has Neon, and the most common x86 architecture has SSE and AVX. We will, however, concentrate on SSE as the most prevalent one. We do not consider AVX because it consumes more power (although, it is difficult to find official numbers) while not providing any noticeable benefits neither to fault tolerance nor to performance.

In the following we discuss in more details the assumptions of our design (including system and fault models), the overall architecture and its limitations.

3.1 Assumptions

We assume a general-purpose microprocessor with support of SSE4.2. This assumption holds on all Intel processors starting from Nehalem microarchitecture and AMD starting from Barcelona or, put simply, on most commodity processors produced after 2007. We also assume no memory or register protection. Even though ECC protection for memory is quite common nowadays, studies show that it does not cover all the faults [SPW09; NDO11]. Therefore, our system model makes no assumptions on memory reliability and admits errors in DRAM and CPU caches.

Our solution requires source code of the program, not an executable binary. It is caused by the fact that it works as a source-to-source compiler (more about it in the next section). Moreover, we rely on the correctness of the code itself and do not protect from software bugs.

We use the Single Event Upset (SEU) fault model, which basically means that only one bit-flip is expected during the whole execution of a program. By a bit-flip, we mean an unexpected change in the state of a memory cell or a CPU register. This also includes errors in memory bus, registers, functional units, etc. The SEU is not permanently damaging the hardware but is transient and lasts only until the next write to the cell/register.

Even though we assume SEU, our approach can work with more than one bit-flip, as long as they do not change both duplicates in the same way. Indeed, by design SIMD-Swift detects all faults except those that alter both copies of the same data. One vivid example of a fault that cannot be detected by our approach (false negative) can be a noise with a high energy level that sets all bits of the register to the same value (all "1" or all "0"). The other potential source of false negatives could be a common mode failure since both duplicates are held in a single register or memory location. Our fault model also does not cover control flow errors, except those caused by a fault in control flow instruction arguments. For example, in statement `if(a)`, faults will be caught in variable `a`, but not in `if` itself.

Our sphere of replication (SoR) covers the CPU and the memory used by the protected program and we do not include the operating system, disk and network subsystems in it. Our solution considers only *fault detection*, not *fault recovery*, although any recovery mechanism can be applied upon fault detection. However, recovery mechanisms are usually very costly and are applied only in the infrequent cases of errors, while fault detection should be continuous and thus efficient. As such, we adopt a crash-stop model when a program is forced to crash upon a detected fault.

3.2 System and Fault Models

As mentioned above, our approach works as a source-to-source compiler, that is, a program that takes a source code as an input, makes changes in its structure and returns a modified source code. Specifically in our case, the functionality is not changed but the fault detection capabilities are added to the application — it continuously checks its own integrity.

Another way for implementing our approach could be a compiler extension (such as an LLVM pass), which changes code at compile-time. This would give a benefit of working with a code that has already been optimized by a compiler and thus, it might improve performance of our solution. It, however, would require much more time for development and we consider it as our future work. That is why the source-to-source compiler approach was chosen for a prove-of-concept implementation.

Before we dive deeper into the architecture, consider a general structure of a normal compiler, shown in Figure 3.1 [Aik14].

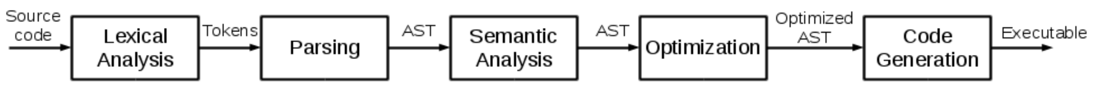

Figure 3.1: Compiler structure

In the stage of the *Lexical Analysis*, program source is divided into so-called "tokens" (words) of the program. After that, the *Parser* tries to define a structure of the program by grouping together tokens into higher level constructs. The result of the parsing stage is the Abstract Syntax Tree (AST), which captures the nesting structure of the program, but abstracts from the concrete syntax. For example, an expression "5 + (2 + 3)" will be transformed into the AST shown in Figure 3.2. The next stage is the *Semantic Analysis* in which the compiler tries to make variable bindings and find inconsistencies like type mismatches, scope violations, etc. The *Optimization* step makes a set of changes to the program structure in order to make it run faster or use less memory, or even consume less power. The *Code Generator* translates the resulting AST into a programing language, usually into the target machine assembly.

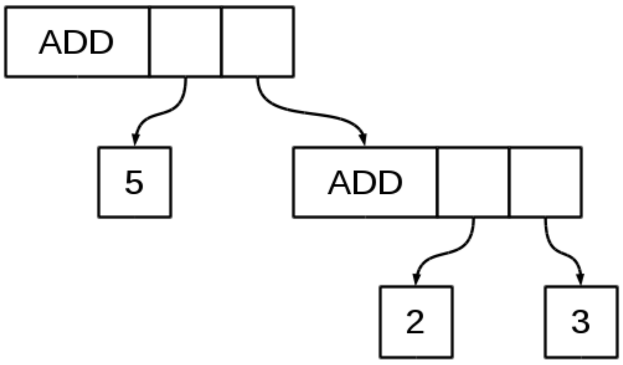

Figure 3.2: Example of Abstract Syntax Tree

Our source-to-source compiler has a slightly different structure. In the first place, it has a weaker semantic analysis and no optimizations, because the C compiler (e.g., GCC) will perform these operations on the resulting code anyway. The second major difference is a new *Transformation* stage, which actually applies SIMD-Swift to the target program. Figure 3.3 shows a resulting structure of our framework.

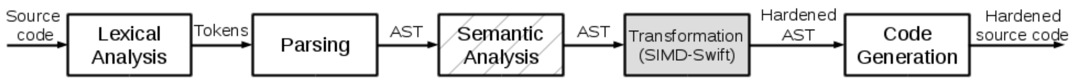

Figure 3.3: SIMD-Swift compiler structure

Consider a small example shown in Listing 3.1.

```
        // b, c and d are input arguments
        a = b + c;
        r = d * a;
        printf("Result: %d", r);
```
Listing 3.1: Original target code

The transformer makes the following set of changes to the program:

Variable conversion. In order to use a hardened code in a non-hardened environment we need an entry point. We use a `computation` function for this purpose, which duplicates input arguments, makes a call to the hardened part of the program using duplicated arguments and then extracts the result from the return value. Figure 3.4 shows a process of duplication and extraction. Here XMM — SSE registers, GPR — general-purpose registers.

All variables inside of the hardened part of the program are replaced by duplicated versions.

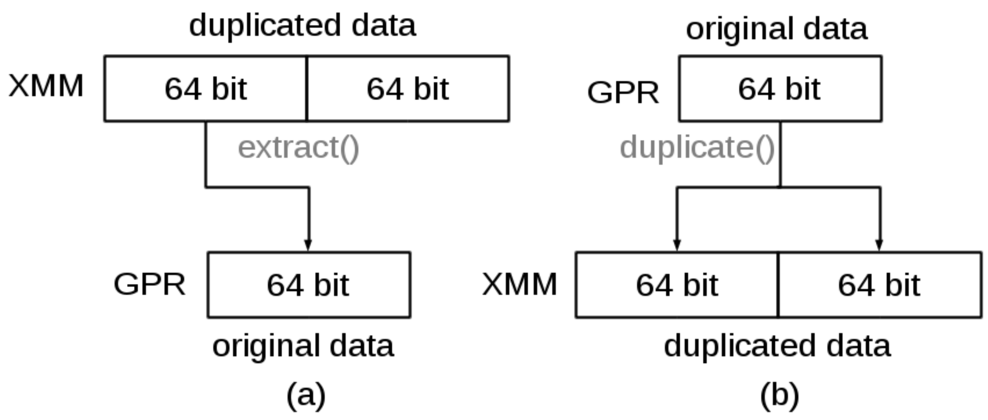

Figure 3.4: Variable conversion
(a) Extraction, (b) Duplication

After this set of transformations our target code will be changed to Listing 3.2.

```
        // b, c and d are input arguments
        b_dup = duplicate(b);
        c_dup = duplicate(c);
        d_dup = duplicate(d);

        // computation over hardened variables
        a_dup = b_dup + c_dup;
        r_dup = d_dup * a_dup;

        // get result
        r = extract(r_dup);
        printf("Result: %d", r);
```
Listing 3.2: Target code with variables converted

Checks. Before the program returns the output variables, they need to be checked for integrity. It should also be done in all points that may influence the output of the program, but we use accumulators there instead (see below).

In our case, the check is a comparison of two duplicates for equality. Indeed, such comparison definitely detects a fault, since the fault can change the output value in only one copy of the result (by assumption in Section 3.1). This can be challenging, since we have to compare two parts of the same register and, as we show in Chapter 4, modern SIMD implementations do not provide a single dedicated instruction for that. To guarantee that all program results were indeed correctly computed, all output variables of the program are checked.

In our example, a check should be added before the `printf` statement, which acts as an output event (see Listing 3.3, line 11).

Accumulators. The check operation is usually expensive (see Chapter 4) and would harm performance if used everywhere in the program. In order to avoid this, we applied checks only to the outputs of the program, and in all points that may influence an output (memory access and control flow operations) we used *accumulation*. Such replacement is valid since the *accumulator* is a variable that is dependent on all critical variables in the program, which means that an error in any of these variables will lead to an error in the accumulator. Check of the accumulator in the end of the program shows if any error appeared during its execution. Any operation that satisfies this requirement can be used as an *accumulation* — for instance, addition. As an example, if `a` and `b` are the critical variables, and a fault affected `b`, than `accum = a + b'` will be also affected, and the check on `accum` will signal an error.

Moreover, there is no need to check or accumulate the results of all instructions because most of the variables are dependent on each other, that is, an error in one of them will propagate to the others. For example, an error on line 2 of Listing 3.1 will lead to a wrong result of the multiplication on line 3 and will propagate further to the `printf` statement. It means that it is enough to accumulate only the resulting value printed on line 3.

Accumulation is required on the following critical operations of the program:

- *Memory accesses (loads and stores)*. If a memory address contains an error, it will lead to a load of a wrong value. This category includes pointer dereference, array and structure element access.

- *Control flow operations*. An error in variables that are used for branching may lead to a change in the control flow direction. For example, an error in branch condition can lead to taking a `False` branch when it had to be a `True` branch. It includes `if` and `switch` statements, and conditions in loops.

Basic and composite operations. By basic operations we understand operations that cannot be further subdivided into components. Such operations are replaced by ad-hoc wrappers. For example, the addition `a = b + c` will be replaced by the function call `a = add_enc(b, c)` (see Listing 3.3). Composite operations that consist of basic ones are replaced by subsequent calls to wrappers, with casts and conversions added if required. For example, `d = a + b - c` transforms into `d = sub(add(a, b), c)`.

```
    // b, c and d are input arguments
    b_dup = duplicate(b);
    c_dup = duplicate(c);
    d_dup = duplicate(d);

    // basic operations replaced with hardened versions
    a_dup = add_enc(b_dup , c_dup);
    r_dup = mul_enc(d_dup , a_dup);

    // get result
    check(r_dup);
    r = extract(r_dup);
    printf("Result: %d", r);
```

Listing 3.3: Target code with basic operations replaced by wrappers

Comparisons. Comparisons in SSE implement a different interface than normal ones. Usually, comparisons toggle the status register (EFLAGS in x86) to affect the control flow of a program. In SSE, however, all comparisons return an integer value with either all "1" in case of `True`, or all "0" in case of `False`. Any other value will indicate an error, because it is impossible (in error-free case) that two copies of the same data produce different comparison results. For example, if we have duplicated variables `a` and `b`, it is impossible for one copy of `a` to be greater than `b` and another — to be smaller. If it is actually the case, this indicates a fault.

To actually influence the control flow of the program, the resulting value is converted into a boolean just before usage in control flow statement. For example, a simple branch in Listing 3.4 will be transformed into Listing 3.5.

```


if (a < b) {
  // do something
}
```

Listing 3.4: Original branch

```
int128 c = less_than(a, b);
int c_bool = to_boolean(c);
if (c_bool)
  // do something
}
```

Listing 3.5: Hardened branch

Library calls. Function calls to external libraries cannot be hardened since we do not have access to their source code. That is why we accumulate the arguments of the call to ensure correct input, execute the non-protected call and then duplicate the result.

Consider a small example of the source code transformation performed by the SIMD-Swift source-to-source compiler. Listings 3.6 and 3.7 show the comparison of the original code with its hardened version. The code is the simple function that calculates a given element of the Fibonacci sequence.

```
 1
 2
 3 int computation(int n) {
 4   int i, next;
 5   int first, second;
 6   first = 0;
 7   second = 1;
 8   i = 2;
 9   for(; i < n; i++) {
10
11
12
13
14
15
16
17     next = first + second;
18     first = second;
19     second = next;
20
21   }
22
23   return next;
24 }
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41 int main() {
42   int n = 20;
43   int next = computation(n);
44   printf("Result: %d", next);
45   return 0;
46 }
```

Listing 3.6: Original code

```
 1 int128 accum = 0;
 2
 3 int128 computation_enc(int128 n) {
 4   int128 i, next;
 5   int128 first, second;
 6   first = to_int128(0);
 7   second = to_int128(1);
 8   i = to_int128(2);
 9   for(;;) {
10     accum += i;
11     accum += n;
12     // note inverse condition
13     if(extract(geq(i, n))) {
14       break;
15     }
16
17     next = add_enc(first, second);
18     first = second;
19     second = next;
20     i = add_enc(i, to_int128(1));
21   }
22
23   return next;
24 }
25
26 // wrapper for old computation
27 int computation(int n) {
28   int result;
29   int128 n_dup, result_dup;
30
31   n_dup = to_int128(n);
32   result_dup = computation_enc(n_dup);
33   if !check(result_dup) || !check(accum)
34     crash();
35   result = extract(result_dup);
36
37   return result;
38 }
39
40 // left as-is
41 int main() {
42   int n = 20;
43   int next = computation(n);
44   printf("Result: %d₩n", next);
45   return 0;
46 }
```

Listing 3.7: Hardened code

The `main` function represents an entry point to the hardened part of the program — `computation` function, which is transformed using SIMD-Swift.

After the transformation, the `computation` function does not perform its main functionality anymore and becomes a point of data conversion. On lines 28–29, we initialize variables and on line 29 we use a pseudo-type `int128` which represents a duplicated integer. On line 31 we convert an input argument to this type and pass it to the hardened `computation_enc` function on line 32. After that, the result and the accumulator are checked for errors on line 33 and decoded back to integer on line 35.

All the original functionality is moved to the `computation_enc` function on lines 3–24. The main change here is the loop (see Listings 3.8 and 3.9). It is broken into three pieces: the loop itself is replaced by generic infinite `for` on line 1, the loop invariant is pulled out into the separate `if` statement on lines 3–4 and the loop counter incrementation is also pulled in the separate operation in the end of the loop, line 5. This reformatting does not affect the semantics of the loop, but simplifies the analysis and transformation of source code.

```
1 for(; i < n; i++) {
2
3
4
5
6 }
```

Listing 3.8: Original loop

```
1 for(;;) {
2   // note inverse condition
3   if (i >= n)
4     break;
5   i++;
6 }
```

Listing 3.9: Refactored loop

Going back to our example in Listing 3.7, a check on loop invariant is replaced by the wrapper function `geq` on line 13, result of which is immediately decoded to boolean using `extract`. Before that, variables `i` and `n` are accumulated, because an error in each of them may influence control flow of the program. Counter incrementation is also replaced by a wrapper function call and put on line 20. The variables' initialization in the hardened version is done using duplicated constants on lines 6–8.

4 Implementation

In the previous chapter we discussed a general architecture of SIMD-Swift. Now let us consider its implementation details. We will start with technologies used in each step of the source-to-source compilation and then will go into more details of the transformation stage. We will also examine the drawbacks of our implementation.

4.1 Used technologies

As mentioned in Chapter 3, our implementation includes five steps: Lexical Analysis, Parsing, Semantic Analysis, Transformation and Code Generation.

The first step of Lexical Analysis uses Python Lex-Yacc (PLY) library [Bea01], which provides an extensible Python implementation of common tools `lex` and `yacc`, and allows to build lexers and parsers upon it. Even though it is quite slow due to its Python implementation, it can be used for languages with complex grammar rules and allows fast and clear prototyping.

Parsing and Code Generation are implemented using PyCParser [Ben10]. PyCParser is a C language parser, written in pure Python and based upon PLY.

Semantic analysis and Transformation stages are based on our inhouse encoding framework that provides a generic implementation of a source-to-source compiler with extensible hooks for writing concrete encoders. In this thesis we have developed such an extension for SIMD-Swift and in the following, we will consider its essential elements.

4.2 Transformations

The Transformation stage performs three fundamental changes to the program: it changes data types, replaces operations and adds checks and accumulations.

4.2.1 Data types replacement

On the intrinsics level (see Chapter 2.2.2), SSE has only three data types available:

- `__m128` — 4 packed single-precision floating-point variables;
- `__m128d` — 2 packed double-precision floating-point variables;
- `__m128i` — packed integer variables (e.g. 2 64-bit integers);

All basic types are replaced as shown in Table 4.1. We use only two types `__m128d` and `__m128i` containing 2 64-bit integers, since we do not need more than two copies for fault detection (under the assumptions discussed in Chapter 3.1) and it simplifies the implementation. This may lead, however, to changes in a program behavior if, for instance, it relies on 32-bit integer overflow. But we do not address the issue and leave it for future work, assuming that our current benchmarks do not have this behavior.

Original type	Replacement type
Integer types (including int8, uint8, int16, etc.)	__m128i
Pointer types (including arrays and structures)	__m128i
float, double	__m128d
_Bool	__m128i

Table 4.1: Type mapping.

User-defined types and structures are tracked down to basic ones and replaced accordingly. For example, a structure {int x, int y} is replaced by its duplicated version {___m128i x, ___m128i y}. Enumerations are considered a special case and are treated as integer constants.

By replacing the data types, we ensure that the original code is hardened, i.e., all variables become duplicated and all operations are performed on two copies.

4.2.2 Basic operations

By basic operations, we understand operations that cannot be further divided into sub-operations and, in most cases, have a one-to-one mapping to assembly instructions (e.g. add, mul, xor, etc). Since we use two data types — __m128i and __m128d, all such operations must have two corresponding implementations, one for 64-bit integers and one for double-precision floating-points (doubles).

Arithmetic operations.

Most arithmetic operations have an implementation in SSE for both integer and floating-point types, since SSE was initially targeted for data processing. Issues appear only with integer division and modulo operations which are used quite rarely and hence have no SSE equivalents. Since they cannot be replaced with any other operations (at least not in a general case), we had to use an EDDI-like approach [OSM02] and execute them twice. Table 4.2 shows instructions used for arithmetic operations.

Operation	Assembly (int)	Assembly (float)	Intrinsic (int)	Intrinsic (float)
Addition	PADDQ	ADDPD	_mm_add_epi64	_mm_add_pd
Subtraction	PSUBQ	SUBPD	_mm_sub_epi64	_mm_sub_pd
Multiplication	PMULQ	MULPD	_mm_mul_epi64	_mm_mul_pd
Division	IDIV x2	DIVPD	Duplicated	_mm_div_pd
Modulo	IDIV x2	-	Duplicated	-

Table 4.2: Arithmetic operations.

Division and modulo could also be implemented in the following way: we could convert integer values to floating-point, perform floating-point division/modulo and then convert back. It would have, however, even higher overhead since SSE conversions are expensive (e.g., conversion from `__m128i` to `__m128d` takes 4 cycles on Intel Haswell) and cannot be executed in parallel due to data dependency.

Operations with pointers.

Pointers duplicated in `__m128i` cannot be used directly because modern processors require a memory address operand to be a 64-bit integer. That means before each memory access, we need to extract a pointer and accumulate its duplicated value, since we use only one copy and an error in this copy may silently lead to a load of a wrong value or a store to a wrong address (see Figure 4.1).

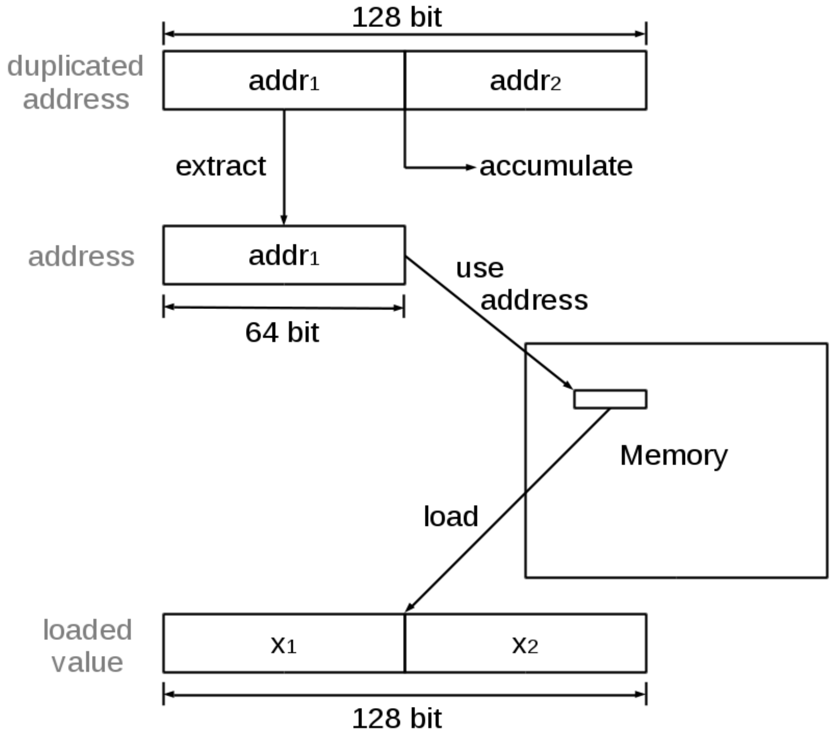

Figure 4.1: Memory access.

Pointer arithmetic (i.e., adding an integer to a pointer or subtraction of two pointers), however, may be performed without these additional operations. We can execute all operations directly on duplicated values and extract lazily — right before the memory access.

Boolean logical operations.

Boolean logic has no implementation in SSE, but it can easily be replaced by other operations. Boolean AND (&& in C) can be replaced by integer multiplication since they have the equivalent behavior with boolean numbers (`False` — zero value, `True` — non-zero). Table 4.3 shows the truth table of both multiplication and logical AND.

Logical OR (|| in C) can be replaced by a bitwise version. In bitwise OR, if at least one of two variables is not equal to zero (contains bits with value "1"), then result will also

Operation	Resulting value
True && True	!= 0
True && False	0
False && False	0

Table 4.3: Truth table for logical AND.

be non-zero — which is a behavior of logical OR. Table 4.4 illustrates this on two cases: both variables are zero (False) and at least one variable is non-zero (True).

True OR True	False OR False
0100	0000
0010	0000
0110	0000

Table 4.4: Logical OR replaced by bitwise OR.

Bitwise logical operations and shifts.
Most of the bitwise logical operations are available for 128-bit variables in SSE as shown in Table 4.5. The only exception is NOT (logical complement), but it can be replaced with XOR with a register containing all bits set to "1".

Operation	Intrinsic	Assembly
AND	_mm_and_si128	PAND
OR	_mm_or_si128	POR
XOR	_mm_xor_si128	PXOR
Left shift	_mm_sll_epi64	PSLLQ
Right shift	_mm_srl_epi64	PSRLQ

Table 4.5: Bitwise operations.

Comparisons.
SSE includes all floating point comparisons, but only Equal (PCMPEQQ) and Greater Than (PCMPGTQ) are implemented for integers. All other operations, however, can be replaced with those two as presented in Table 4.6.

4.2.3 Composite operations

A general approach to encoding any composite node in Abstract Syntax Tree (AST) is to encode the node name (in order to avoid conflicts with non-hardened variables), replace types and recursively encode child nodes until we reach basic operations. This way there is no need in implementing composite operations since we already have a way to encode basic ones and all their combinations.

Operation	Replacement
a != b	(double) a != (double) b
a < b	b > a
a <= b	!(a > b)
a >= b	!(b > a)

Table 4.6: Implementation of comparisons.

4.2.4 Integrity checks

In the critical points of execution duplicated variables have to be compared for equality in order to avoid error propagation (see Chapter 3.2). Such operation is not available in SSE, but it can be replaced with horizontal subtraction, that is, a subtraction of scalar variables inside of a packed variable. If two duplicates are equal, the result will be zero. If there was an error and the duplicates are different, the result of their subtraction will be non-zero. Afterwards, we can check this result with PTEST that performs a test for all zeros and returns a boolean value. This value, in turn, can be used for branching and executing a recovery mechanism in case of error. Since we operate on 64-bit integers and horizontal subtraction is available only for 32-bit integer and smaller, a shuffle has to be performed before subtraction to reorder 32-bit parts of a register. Figure 4.2 illustrates this idea. Unfortunately, this sequence of instructions has a total latency of 6 cycles (on Intel Haswell) and therefore is quite expensive.

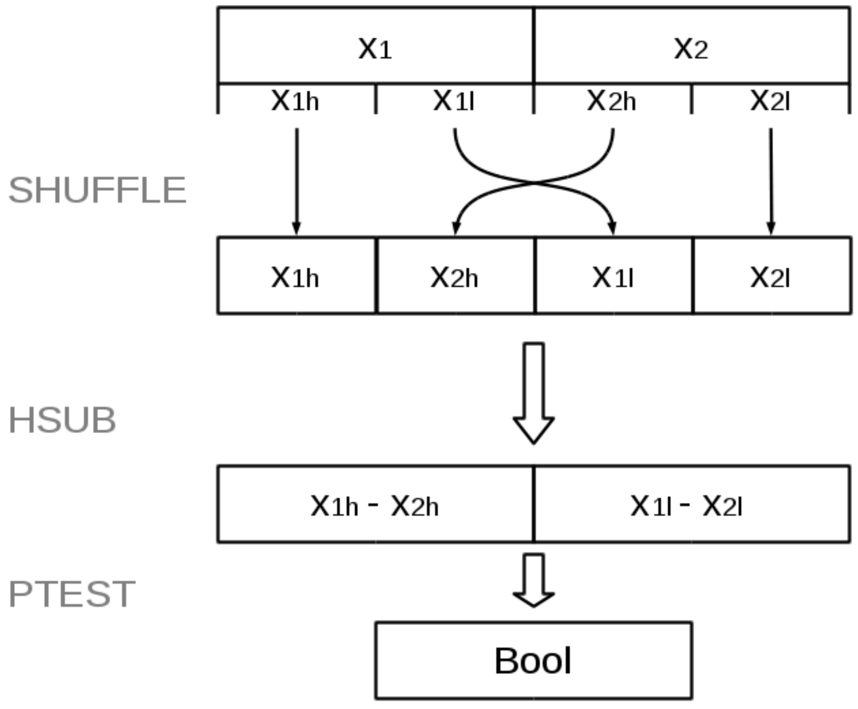

Figure 4.2: Implementation of checks.

Accumulations.
Accumulation is much simpler and thus faster. It is implemented with the addition (PADD) of a variable-to-check to an accumulator. In order to avoid redundant interactions with main memory, one of the XMM registers is dedicated to accumulation (e.g., by using `register` keyword in gcc), which leads to a latency of only 1 cycle (on Intel Haswell).

4.3 Performance bottlenecks

Our approach has two major performance bottlenecks: comparisons and memory accesses. Both of them are caused by the non-availability of corresponding SSE instructions and thus, an increased number of instructions is required to perform the operation.

4.3.1 Comparisons

The main issue of comparisons in SSE is that they do not set the EFLAGS register and therefore, cannot be used for control flow operations (branching) directly. In order to use them, we need an additional instruction that converts the resulting _m128i value to a boolean and sets the corresponding flags. Moreover, we require accumulation before jumps since their arguments influence a control flow of a program. Listing 4.2 shows an example of additional instructions that appear after hardening.

```

; implicitly set flags in EFLAGS
CMP %eax, %ebx








; take a branch based on set flags
JGT .L0
```

Listing 4.1: Original comparison (2 instructions — 2 cycles)

```
; put 0's or 1's in XMM1
; for later check
PCMPGTQ %xmm0, %xmm1

; accumulate in register xmm15
PADDQ    %xmm1, %xmm15

; AND with all 1's register xmm14
; and set ZF flag if result
; is zero
PTEST   %xmm14, %xmm1
JE      .L0
```

Listing 4.2: Hardened comparison (4 instructions — 6 cycles)

4.3.2 Memory accesses

This bottleneck is similar to comparisons. A memory access requires an address operand to be a 64-bit integer, but pointers in a hardened program have type __m128i. That means we need an additional extraction, and also accumulation since an error in the address will either lead to a load of wrong value or a store to the wrong address (lost update). Listing 4.4 shows such transformation.

```
; load from address
; specified in %rax
; a value into %ebx
MOV      (%rax), %ebx
```

Listing 4.3: Original memory access (1 instruction — 1 cycle)

```
; xmm0 contains address operand.
; Accumulate it in register xmm15
; for later check
PADDQ    %xmm0, %xmm15

; extract address
MOVQ     %xmm0, %rax

; use address to load value
MOVAPS   (%rax), %xmm1
```

Listing 4.4: Hardened memory access (3 instructions — 6 cycles)

Situation is even sadder than with comparisons, since SSE moves often have a high latency (e.g., AMD Piledriver requires 5 cycles for each movement between XMM registers). It gets even worse when we access an array element, because we also need to extract and accumulate the element's index.

4.4 Alternative implementation

We can avoid the memory access bottleneck by not duplicating pointers at all. This way we trade fault tolerance for performance by having less computational overhead but at the same time making our approach vulnerable to errors in pointers. Let us consider consequences of applying such a trade-off.

An error in pointer that is used for loading may lead to the following consequences:

1. The address becomes unreachable. In this case, a segmentation fault will be thrown and the program will crash (benign failure).
2. The address points to allocated but unused memory. The error will lead to Silent Data Corruption (SDC) because allocated memory is initialized with zeros and two equal zero-filled duplicates will be loaded.
3. The address points to unallocated or to allocated and used memory. The error will be detected if a first loaded duplicate is not equal to the second one. This situation is most probable since this memory address will contain random values and two subsequent 64-bit chunks of memory are likely to be different.

If a pointer is used for storing, it will lead to a benign crash in case (1) and to a lost update in all the other cases.

Our hypothesis is that most errors in pointers will lead to a segmentation fault and those that will not, will be caught with high probability by SIMD-Swift as shown in case (3). In order to verify it we designed two tests.

In the first test, we load a known integer value from memory and then check if it is correct (original version). In the second test, we do the same, but the value is duplicated and the duplicates are checked for equality after the load (hardened version). The original version can catch errors of case (1), while the hardened version also catches errors of case (3). In both tests our goal is to define what is a probability of SDC. In order to do this, we inject a single-bit flip fault in the load address and check the resulting value. In the beginning of each test we allocate a chunk of memory, which allows us to see the impact of allocated-memory size on the SDC rate. Figure 4.3 shows the results of these tests. Here, "hardened" means a version where pointer is not duplicated but memory value is. We increase the size of allocated memory only up to 800 MiB because it is the highest size we could allocate on a Linux machine with 2GB of RAM.

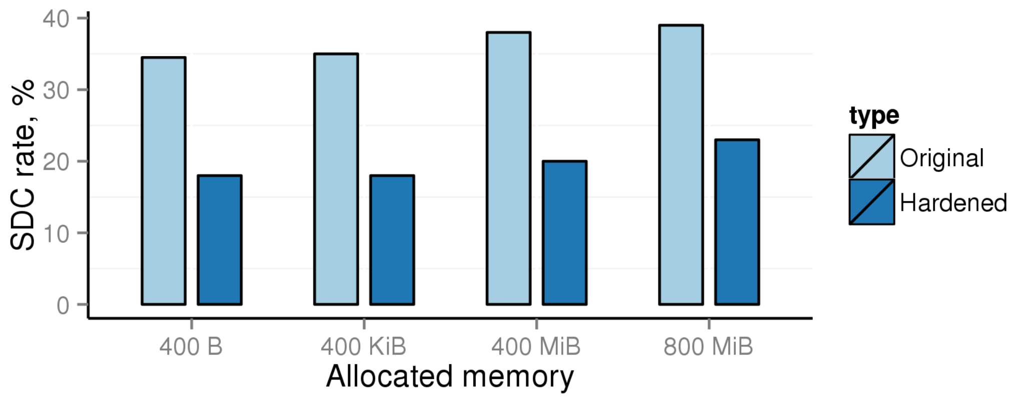

Figure 4.3: Results of pointer fault injection.

As we can see, only in 35–39% of cases an error in a pointer will lead to SDC even for non-hardened program, and it gets lower with SIMD-Swift because some errors are caught as in case 3. These results give us hope that such an alternative implementation may not cause a significant decrease in fault detection capabilities while giving us an improvement in performance.

Also, we can notice a slight correlation between the SCD rate and the size of allocated memory, but it is insignificant.

An evaluation of performance overhead and fault tolerance for both full and alternative implementations will be presented in Chapter 5.

5 Evaluation

In this chapter, we will consider fault detection capabilities and performance overhead of SIMD-Swift for both the full implementation (referred to as “full”) and the implementation without pointer duplication (referred to as “pointerless”). The set of tested benchmarks consists of:

- basic integer algorithms: Bubble sort, Fibonacci sequence and Sieve of Eratosthenes (SoE);
- floating-point algorithms: numerical implementation of sine and Fast Fourier Transform (FFT);
- benchmarks from MiBench [Gut+01]: Dijkstra’s algorithm and Cyclic Redundancy Check (CRC);

Bubblesort and Dijkstra represent memory-access-dominated algorithms. FFT and Sine consist primarily of arithmetic instructions. SoE, CRC and Fibbonacci contain a significant amount of control-flow operations.

We will start with the performance testing, continue with the fault injection and then discuss the results.

5.1 Performance testing

For the performance overhead evaluation we used a computer with the following characteristics:

- RAM: 8GB
- CPU: Intel Core i5-5200U (Broadwell microarchitecture)
- Caches:
  - L1: 128 KB
  - L2: 512 KB
  - L3: 3072 KB

The benchmarks were compiled using the GCC version 5.1.1 with SSE4.2 enabled (`-msse4.2`), AVX disabled (`-mno-avx`), x86-64 instruction set (`-m64`) and all optimizations (`-O3`). For the performance testing we used PyCPerf tool [Ole15] which uses the Time Stamp Counter and RDTSC instruction for measuring how many CPU cycles it takes to execute a given region of code. All benchmarks were running for at least one second with the fixed input. Each benchmark was tested in two variants: “full” — all variables are duplicated, and “pointerless” — pointers are not protected by duplication. Figure 5.1 shows the performance overheads of all benchmarks compared to the native execution.

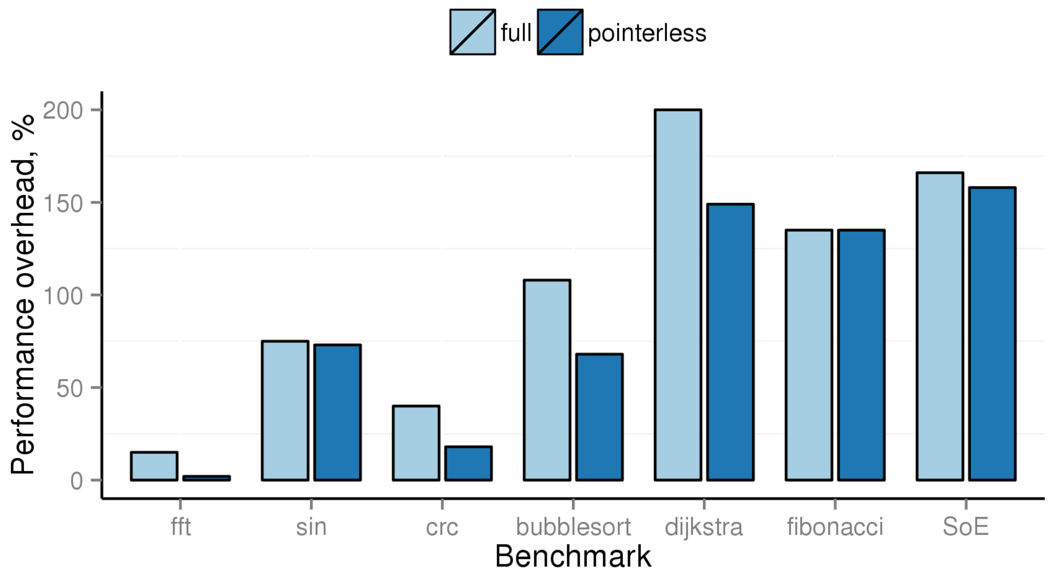

Figure 5.1: Performance testing results.

As we can see, the results are varying for different types of benchmarks. The benchmarks are grouped by performance results: for FFT, CRC, Sine and Bubblesort (pointerless implementation) we managed to get less than 80% overhead, and others perform worse than SWIFT (more than 100% overhead). It is caused by the fact that some benchmarks are influenced by the bottlenecks (see Chapter 4.3), and some are not. We discuss this variance in detail in Section 5.3.

It should be noted that the results may be different not only for other processor manufacturers and models (e.g., AMD) but even for other generations of Intel processors, since every next generation makes a bit different performance trade-offs than the previous one. For example, PMULUDQ (multiply the low unsigned 32-bit integers from each packed 64-bit element in two __m128i variables) has a latency of 3 cycles in Ivy Bridge, whereas the same instruction’s latency in Haswell is 5 cycles. But the general tendency will stay the same.

5.2 Fault injection

For the fault injection campaign, we used Intel Pin [Int04] and BFI [Beh15] tools.

Pin is a binary instrumentation tool, that is, the tool that enables the runtime instrumentation on the compiled binary files. BFI uses this functionality to perform the runtime fault injection. In our case, BFI injects single-bit faults in CPU registers, memory cells and the address bus, once per run (see also our fault model in Section 3.1). The faults are random and uniformly distributed in a region of code under consideration. A fault may result in one of the five broad types of consequences:

- Silent Data Corruption (SDC) — the fault changed the result, but stayed undetected.
- Crash — the fault led to an externally-visible program crash. In most cases, it is a segmentation fault.
- Masked — the fault did not affect the result.
- Detected — SIMD-Swift detected an error.
- Hanged — the fault caused a hang of the program.

The main goal of SIMD-Swift is to reduce the number of SDC, but if the rate of crashes is increased by our solution, we also consider it as an acceptable result, since they are externally-visible. Many injected faults do not lead to changes in outputs (masked) because they affect unused registers and thus do not propagate further. Hangs can happen due to a fault changing a loop variable and usually are infrequent.

We used three representative benchmarks for testing: FFT is dominated by arithmetic floating-point instructions, Bubblesort is pointer-dominated (memory-heavy) and SoE consists in a large part of control-flow operations (branching). Each benchmark was tested in the original version, with full duplication and without pointer duplication. The fault injection results are shown on Figure 5.2.

5.3 Discussion

The performance testing results clearly show us the influence of the two main bottlenecks, discussed in Chapter 4.3.

We achieved the lowest overhead with the FFT benchmark, which consists primarily of arithmetic floating point operations and a small amount of memory accesses, used for reading input and storing results. Such structure caused only 15% overhead in the full version and almost no overhead when we disabled pointer duplication. The similar situation is with the Sine benchmark, but it contains fewer pointer operations — which explains the small difference between the full and pointerless versions, and more control-flow operations, leading to higher overhead in general.

Bublesort and Dijkstra benchmarks represent memory-access-dominated applications. Bubblesort has all array references substituted by direct pointer dereferencing, while

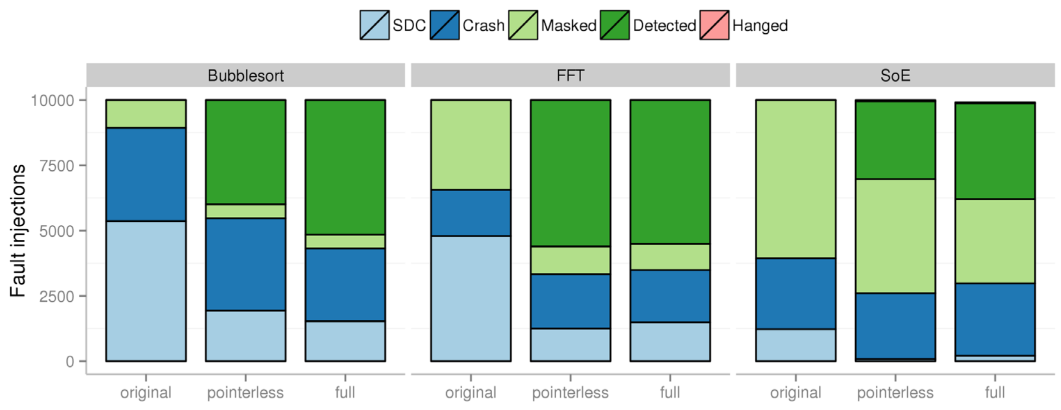

Figure 5.2: Fault injection results.

Dijkstra is written with array references only. As we can see from results, it caused a significant performance improvement for Bubblesort because it extracts address only from the pointer itself, while Dijkstra has to extract both array address and index. Also, we can see that this type of application benefits most from pointerless version, since a significant part of the code becomes non-duplicated.

Fibonacci and SoE, on the other hand, had almost no benefit from the pointerless version, since they are dominated by control-flow instructions and memory accesses constitute only a small portion of their overhead.

CRC stays a bit aside from other benchmarks because its low overhead is caused by numerous calls to the standard library, which are not hardened and thus do not experience any overhead. This highlights an interesting property of SIMD-Swift that it can be applied to only parts of code (for example, some logging and statistics can be left unhardened).

The fault injection campaign shows that a substantial part of faults stays undetected. We examined the sources of SDCs and defined that the vast majority of them are faults in either read or write address. It indicates the main window of vulnerability for SIMD-Swift — a load and store from the memory. Consider a case of reading from the memory (Figure 5.3). In the full version, addresses are kept duplicated all the time, except just before the usage. If a fault occurs in the extracted address, it will be used to load a value from the wrong address, and if the low 64 bits of this value are equal to the high 64 bits, the error will stay undetected and may lead to SDC (see the detailed discussion in Chapter 4.4). The fault injections in SoE benchmark prove this observation, since it contains only a small amount of memory interactions and hence, is less exposed to this window of vulnerability (the SDC rate is roughly 2%).

This vulnerability can be removed by executing all load and store operations twice, that is, by separately working with the first and second duplicates of the pointer. We manually refactored Bubblesort benchmark to use such redundant memory interactions and received only 1.4% SDC rate, which is a significant improvement in comparison to

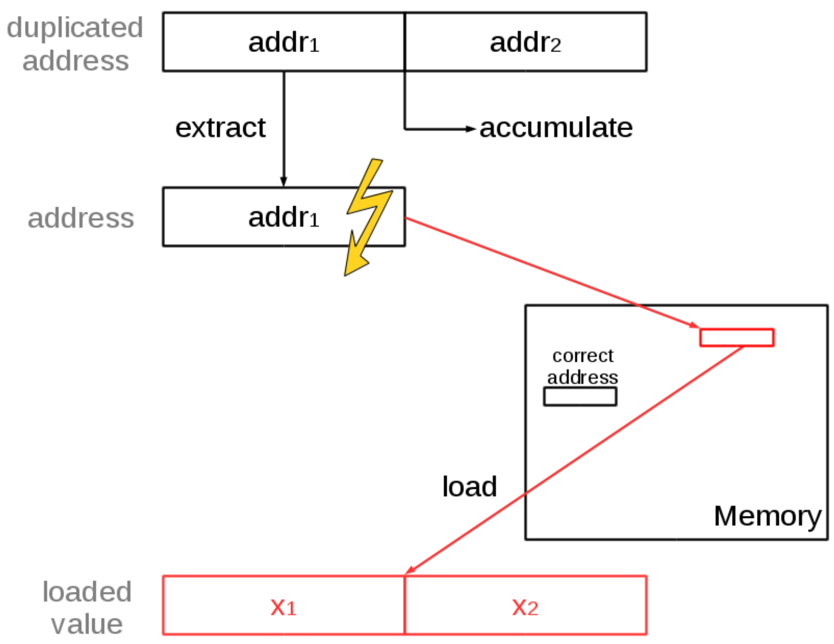

Figure 5.3: Window of vulnerability.

our previous result — 15.3%. But at the same time performance overhead has grown from 108% to 175% and since other benchmarks are expected to have the similar increase, we decided not to implement this version as an unpromising one.

6 Conclusion And Future Work

Modern hardware shows a trend to an increase in fault rates caused by shrinking feature sizes and decreasing voltages. The cheapest way to deal with this issue is to use software-level redundancy, that is, to duplicate some part of a code or some variables. Nowadays, a wide range of such solutions exists, but most of them are lacking one core quality — performance. Therefore, our thesis was targeted to develop a software-based fault tolerance approach which could improve performance results of the current solutions. In order to do that, we used the Single Instruction Multiple Data (SIMD) technology to duplicate all program's variables. Such duplication allows to detect faults by a simple comparison of two copies under the assumption that only one copy is affected by a fault. This idea is based on the observation that SIMD is usually underutilized in modern CPUs. We implemented it as a source-to-source compiler which performs hardening of a program on the source code level.

As we have seen in Chapter 5, we did not achieve a desired result for all benchmarks due to two inherent bottlenecks — memory accesses and comparisons. Although it is still reasonable to use SIMD-Swift for the applications that are dominated by arithmetic or logic operations (e.g., FFT and Sine benchmarks), such applications are not very common. We believe our approach is most well-suited for those floating-point benchmarks that do not benefit from SIMD vectorization. Therefore, in many cases the performance overhead is expected to be higher than with alternative hardening approaches, such as SWIFT.

This effect is slightly reduced in the implementation without pointer duplication. As we can see from the results of the fault injection, it only slightly decreases the fault detection capabilities of the approach, which is caused by the memory address vulnerability discussed above. At the same time, it significantly improves performance of memory-access-dominated applications by removing one of the main bottlenecks.

All our implementations perform only fault detection, not fault recovery. That means recovery mechanisms have to be applied upon fault detection. The implementation without pointer duplication, however, relies on the assumption that crashes (especially, segmentation faults) can be tolerated and used for detection and this, in turn, decreases the range of fault recovery mechanisms which can be used with it. For example, if we use the Triple Modular Redundancy approach for recovery by having three copies instead of two, we will be able to recover only from detected faults and a crash will just stop the execution.

Our main bottlenecks are caused by the fact that memory and control flow operations do not have direct SSE implementation in hardware (see Section 4.3). For current processors this issue is unsolvable, but the next generation of the Intel Xeon processors (Intel Xeon E5-26xx) is expected to have a field-programmable gate array (FPGA) [SPW09], which can be used for implementing such functionality. When the new Xeon processors will go to the market, we may get a significant improvement in performance for the SIMD-Swift approach by adding our own, SIMD-Swift-based instructions. Another issue discovered by the fault injection tests is the window of vulnerability during a memory access, caused by a usage of non-duplicated addresses. It could potentially be resolved by having duplicated load and store operations, but in current hardware it worsens performance even further. We tried this approach on the Bubblesort benchmark and got an increase in overhead from 108% to 170%, which is much higher even than SWIFT. Having FPGA, however, may help in this case too, since it would allow us to implement the duplicated memory access in the hardware. As such, FPGA-assisted approach is a promising avenue for future work.

Our implementation is also lacking a full data type support. In the current version, we cast all data types to two basic ones — packed integer (`__m128i`) and packed double (`__m128d`). In most cases it does not cause any problems, but it may change a program's behavior if it relies on integer overflows. We have made such decision to get a proof-of-concept implementation and evaluate the approach in general. We consider full data type support as our future work.

The source-to-source compiler implementation significantly restricts the complexity of used benchmarks since full support of C requires immense development efforts. That is why we evaluated the approach on this restricted implementation, and in future we are going to reproduce it as an LLVM pass. That will allow us to test SIMD-Swift on real-life applications.

In the end, SIMD-Swift proves its potential for some specific types of applications, achieving as low as 15% overhead while still providing high level of fault coverage. We believe that FPGA-assisted implementation or a smart mix with regular duplicated instructions can provide significant benefits in terms of performance and reliability.

Bibliography

[Aik14] A. Aiken. *Compilers*. 2014. **url**: https://class.coursera.org/compilers-004 (visited on Sept. 10, 2015).

[ARM10] ARM. *Neon*. 2010. **url**: http://www.arm.com/products/processors/technologies/neon.php (visited on Nov. 2, 2015).

[Aus99] T.M. Austin. "DIVA: a reliable substrate for deep submicron microarchitecture design." In: *International Symposium on Microarchitecture*. 1999, pp. 196–207.

[Bea01] D. M. Beazley. *Python Lex-Yacc*. 2001. **url**: https://github.com/dabeaz/ply (visited on Nov. 1, 2015).

[Beh15] D. Behrens. *BFI*. 2015. **url**: https://bitbucket.org/db7/bfi/src (visited on Nov. 1, 2015).

[Ben10] E. Bendersky. *PyCParser*. 2010. **url**: https://github.com/eliben/pycparser (visited on Nov. 1, 2015).

[Ber+05] D. Bernick et al. "NonStop reg; advanced architecture." In: *International Conference on Dependable Systems and Networks*. June 2005, pp. 12–21.

[BM91] M. Brearley and R.B. Moseley. *Electronic braking system*. US Patent 5,004,299. Apr. 1991. **url**: https://www.google.com/patents/US5004299.

[BS04] W. Bartlett and L. Spainhower. "Commercial Fault Tolerance: A Tale of Two Systems." In: *IEEE Transactions on Dependable and Secure Computing*. Vol. 1. Jan. 2004, pp. 87–96.

[Fen+10] Shuguang Feng et al. "Shoestring: Probabilistic Soft Error Reliability on the Cheap." In: *Symposium on Programming Language Issues In Software Systems*. Vol. 45. Mar. 2010, pp. 385–396.

[Fet+06] E.S. Fetzer et al. "The Parity protected, multithreaded register files on the 90-nm itanium microprocessor." In: *IEEE journal of Solid-State Circuits*. Vol. 41. Jan. 2006, pp. 246–255.

[Gol+06] O. Goloubeva et al. *Software-implemented hardware fault tolerance*. Springer Science & Business Media, 2006.

[Gom+03] M. Gomaa et al. "Transient-fault recovery for chip multiprocessors." In: *International Symposium on Computer Architecture*. June 2003, pp. 98–109.

[Gut+01] M. R. Guthaus et al. "MiBench: A Free, Commercially Representative Embedded Benchmark Suite." In: *IEEE International Workshop of the Workload Characterization*. 2001, pp. 3–14.

[Gwe96] L. Gwennap. “Digital, MIPS Add Multimedia Extensions.” In: *Microprocessor Forum*. Vol. 10. 1996, pp. 24–28.

[Gwe98] L. Gwennap. “Altivec vectorizes powerPC.” In: *Microprocessor Forum*. Vol. 12. 1998, pp. 1–6.

[HB92] J.G. Holm and P. Banerjee. “Low Cost Concurrent Error Detection in a VLIW Architecture Using Replicated Instructions.” In: *International Conference on Parallel Processing*. 1992, pp. 192–195.

[Int04] Intel. *Pin home page*. 2004. **url**: https://software.intel.com/en-us/INPROCEEDINGSs/pintool (visited on Oct. 29, 2015).

[Int14] Intel. *Technology and Computing Requirements for Self-Driving Cars*. 2014. **url**: http://www.intel.de/content/www/de/de/automotive/driving-safety-advanced-driver-assistance-systems-self-driving-technology-paper.html (visited on Aug. 20, 2015).

[KAO05] P. Kongetira, K. Aingaran, and K. Olukotun. “Niagara: a 32-way multithreaded Sparc processor.” In: *IEEE Micro*. Vol. 25. Mar. 2005, pp. 21–29.

[Koh+95] L. Kohn et al. “The visual instruction set (VIS) in UltraSPARC.” In: *Compcon ’95. Technologies for the Information Superhighway*. Mar. 1995, pp. 462–469.

[MBS07] A. Meixner, M.E. Bauer, and D.J. Sorin. “Argus: Low-Cost, Comprehensive Error Detection in Simple Cores.” In: *IEEE/ACM International Symposium on Microarchitecture*. Dec. 2007, pp. 210–222.

[McE81] D. McEvoy. “The Architecture of Tandem’s NonStop System.” In: *Proceedings of the ACM ’81 conference*. 1981, pp. 245–247.

[MKR02] S.S. Mukherjee, M. Kontz, and S.K. Reinhardt. “Detailed design and evaluation of redundant multi-threading alternatives.” In: *International Symposium on Computer Architecture*. 2002, pp. 99–110.

[MM88] A. Mahmood and E.J. McCluskey. “Concurrent error detection using watchdog processors-a survey.” In: *IEEE Transactions on Computers*. Vol. 37. Feb. 1988, pp. 160–174.

[NAS00] NASA. *John F. Kennedy Space Center — Frequently Asked Questions*. 2000. **url**: http://science.ksc.nasa.gov/pao/faq/faqanswers.htm (visited on Sept. 2, 2015).

[NDO11] E.B. Nightingale, J.R. Douceur, and V. Orgovan. “Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs.” In: *EuroSys*. ACM, Apr. 2011.

[Ole15] O. Oleksenko. *PyCPerf*. 2015. **url**: https://github.com/OleksiiOleksenko/PyCPerf (visited on Nov. 1, 2015).

[OSM02] N. Oh, P.P. Shirvani, and E.J. McCluskey. “Error detection by duplicated instructions in super-scalar processors.” In: *IEEE Transactions on Reliability*. Vol. 51. Mar. 2002, pp. 63–75.

[PW96] A. Peleg and U. Weiser. “MMX technology extension to the Intel architecture.” In: *IEEE Micro*. Vol. 16. Aug. 1996, pp. 42–50.

[Ram12] R.N. Ramamurthi. “Dynamic trace-based analysis of vectorization potential of programs.” MA thesis. The Ohio State University, 2012.

[Reb+01] M. Rebaudengo et al. “A source-to-source compiler for generating dependable software.” In: *IEEE International Workshop on Source Code Analysis and Manipulation*. 2001, pp. 33–42.

[Rei+05] G.A. Reis et al. “SWIFT: software implemented fault tolerance.” In: *International Symposium on Code Generation and Optimization*. Mar. 2005, pp. 243–254.

[Rot99] E. Rotenberg. “AR-SMT: a microarchitectural approach to fault tolerance in microprocessors.” In: *International Symposium on Fault-Tolerant Computing*. June 1999, pp. 84–91.

[RPK00] S.K. Raman, V. Pentkovski, and J. Keshava. “Implementing streaming SIMD extensions on the Pentium III processor.” In: *IEEE micro*. 4. 2000, pp. 47–57.

[RR07] V. Reddy and E. Rotenberg. “Inherent Time Redundancy (ITR): Using Program Repetition for Low-Overhead Fault Tolerance.” In: *IEEE/IFIP International Conference on Dependable Systems and Networks*. June 2007, pp. 307–316.

[Shy+09] A. Shye et al. “PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures.” In: *IEEE Transactions on Dependable and Secure Computing*. Vol. 6. Apr. 2009, pp. 135–148.

[Skl76] J.R. Sklaroff. “Redundancy Management Technique for Space Shuttle Computers.” In: *IBM journal of Research and Development*. Feb. 1976, pp. 20–28.

[Sle+99] T.J. Slegel et al. “IBM’s S/390 G5 microprocessor design.” In: *IEEE Micro*. Vol. 19. Mar. 1999, pp. 12–23.

[Smo+04] J.C. Smolens et al. “Efficient Resource Sharing in Concurrent Error Detecting Superscalar Microarchitectures.” In: *International Symposium on Microarchitecture*. Dec. 2004, pp. 257–268.

[SPW09] B. Schroeder, E. Pinheiro, and W. Weber. “DRAM Errors in the Wild: A Large-Scale Field Study.” In: *ACM SIGMETRICS*. 2009.

[Ste15] Steam. *Steam Hardware and Software Survey*. 2015. **url**: `http://store.steampowered.com/hwsurvey` (visited on Aug. 29, 2015).

[WCS09] P. M. Wells, C. Chakraborty, and G. S. Sohi. “Mixed-mode Multicore Reliability.” In: *ACM Symposium on Programming Language Issues In Software Systems*. Vol. 44. Mar. 2009, pp. 169–180.

[Yeh96] Y.C. Yeh. “Triple-triple redundant 777 primary flight computer.” In: *Aerospace Applications Conference*. Vol. 1. Feb. 1996, pp. 293–307.

[YGS09] J. Yu, M.J. Garzaran, and M. Snir. "ESoftCheck: Removal of Non-vital Checks for Fault Tolerance." In: *International Symposium on Code Generation and Optimization*. Mar. 2009, pp. 35–46.

[Zha+12] Y. Zhang et al. "Runtime Asynchronous Fault Tolerance via Speculation." In: *International Symposium on Code Generation and Optimization*. 2012, pp. 145–154.