Dynamic Binary Translation for Optimized Functional Units

A Senior Project in Electrical Engineering & Computer Science

Eli Baum
advisor: Rajit Manohar

May 2019

1 Abstract

Hardware and software systems are often tightly coupled, making changes to one highly reliant on the other. This, in turn, can impede progress in both fields, especially in specialized environments where such changes could be useful. In this project, I propose a combined hardware-software interface, Dynamic Binary Translation for Optimized Functional Units, that attempts to solve this problem. An LLVM pass, operating either at compile- or run-time (via a JIT), replaces certain function calls with optimized hardware circuits, communicating via memory-mapped IO. While initial results do not indicate a definitive performance boost, this framework is ripe for expansion and further optimization, and could eventually outperform pure software implementations. This project also serves as a proof-of-concept for more general dynamic binary translation schemes which allow for seamless execution of programs on different architectures without the need for recompilation.

2 Introduction

Adding new hardware functionality to CPUs requires corresponding updates to compilers, and the subsequent recompilation of relevant code. Understandably, this poses an impediment to the mainstream design of specialized hardware. Chip architects can be slow to innovate, and enterprising software developers may have difficulty taking advantage of new features without resorting to low-level programming. (Consider, for example, the model of many embedded devices, which use memory-mapped registers to control the device. While incredibly flexible, allowing for significant hardware variety with identical instruction sets, this paradigm makes for confusing, hard-to-write, and not particularly portable code.) Thus, we are by and large left with a homogeneous computing landscape where hardware improvements, when they happen, are costly and focused on niche applications.

This proposed architecture allows for hardware extensions to be implemented without modifying the CPU's instruction set. Together with a cross-platform just-in-time (JIT) compiler, compiled programs can be optimized at runtime to take advantage of these hardware units. In a way, DBTOFU is similar to plug-and-play peripherals, but exists on-chip and has minimal driver requirements.

In this research, *gem5*, a computer architecture simulator, was used to model a partial hardware implementation of BLAS (the Basic Linear Algebra Subroutines) on an x86 system. BLAS is a linear algebra package[1] providing common operations on vectors and matrices; it underpins most higher-level modern numerical computation packages.

The following subset of the BLAS Level 1 (vector-vector operations) subroutines was implemented. Each function comes in multiple flavors (complex numbers, double-precision, etc.); while DBTOFU handles only the double-precision real case (denoted by the `d` prefix), extensions to the other cases would be trivial. Below, X and Y are vectors; a is a scalar.

- **scal** vector scale, $X := aX$.
- **axpy** vector add, $Y := aX + Y$.
- **dot** dot product, $X \cdot Y$.
- **nrm2** Euclidean (L2) norm, $||X||_2$.
- **asum** sum of absolute values, $\sum_{x \in X} |x|$.

DBTOFU is further motivated by the so-called 90/10 rule of. In the most common formulation, the rule states that programs tend to spend about 90% of their time executing about 10% of their code. Together with Amdahl's Law, this suggests focusing optimization efforts on small parts of programs is a reasonable endeavor. For numerical computation-heavy programs, for example, implementing BLAS in hardware could lead to significant gains in speed and memory usage.

3 Architecture Description

3.1 Hardware

My first attempt at implementing BLAS in hardware was to directly specify the circuitry through ACT. While this would have been fruitful had testing occurred on an FPGA or physical chip, an ACT simulation would not have been particularly easy to integrate into gem5.

Instead, I described the hardware at a slightly higher level, by creating C++ classes to represent different circuit elements. This, of course, produces a less

[1]Technically, it is a specification for implementing such a package, not the actual implementation itself, which would instead be finely-tuned for each specific system. I used a C implementation called OpenBLAS for benchmarking: `https://www.openblas.net/`

accurate simulation, but worked better with gem5 (since data at that point had already entered the “simulation” rather than “hardware” domain).

DBTOFU is largely implemented with *multiply-accumulate* (MACC) units. Such a unit performs the scalar operation

$$a := a + b \times c$$

This specific operation was chosen for its relevance to linear algebra, and efficiency,[2] but has no bearing on the working of the system at large; for other domains, any circuit could be used in its place.

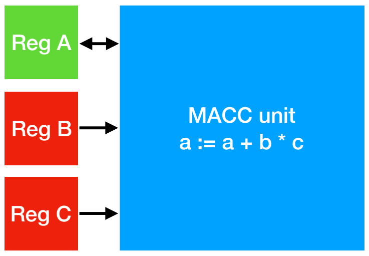

Figure 1: MACC Unit diagram

Limited other arithmetic operations were necessary beyond the MACC units. Latency estimates were established for each of the basic operations (for example, register accesses and MACC execution) and passed back to the simulator after computation was finished. gem5 uses an event-driven timing system, so rather than worrying about simulating each operation’s individual delay, gem5 was simply instructed, before completing execution, to delay for the total amount of time the computation would have taken. This is admittedly less accurate (for example, it ignores the possibility of CPU execution continuing while DBTOFU hardware is running), but easier to simulate.

Communication between software and the functional units was accomplished with memory-mapped I/O (MMIO). One page of physical memory was reserved by the BIOS for the extension. Since the BLAS Level 1 functions have similar function signatures, a common protocol was possible. Treating the mapped page as an array of doubles,[3] the following layout was assigned:

[0] Function ID. Each recognized function is assigned an ID number. Writing to this element tells the FU which operation was requested, and begins

[2] A single MACC unit can often run faster than a multiply followed by an add.

[3] Ideally, all fields would not be doubles. The Function ID and N fields, for example, are integers, and other extensions of this system might want to use other datatypes. However, a homogeneous array was the easiest to implement for a number of reasons. Furthermore, double-to-int casting should not pose an issue, since 64-bit doubles can precisely represent integers up to about 2^{53}.

execution.

[1] Alpha. Scalar argument passed to `dscal` and `daxpy`. Used to return scalars (from all other functions).

[2] N. Length of vector. Since only one page of memory was allocated, and doubles are used throughout, N can be no larger than 254 on the test system.[4]

[3] X array. Continues for N elements, after which point Y array begins.

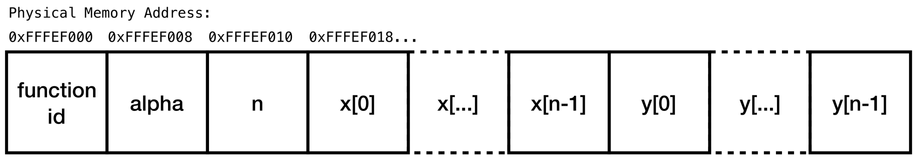

Figure 2: Memory Layout MMIO Communication

3.2 Implementation details of BLAS Level 1 subroutines

For organization purposes, the following common registers were established: rRet, returned values; rS & rR, utility registers for square root computation; rX, X vector; rY, Y vector; rK, constants. While these uses are not strict assignments, hard-wiring a consistent set of registers for each function allows for a more efficient circuit (for example, registers do not have to be reconnected to other MACC units before each computation).

3.2.1 scal

`dscal` performs the operation $X := aX$, where X is a vector and a is a scalar. That this operation can be computed with a MACC is not immediately apparent, but it can be refactored as

$$X := aX \equiv X + (a-1)X$$

This is now in "MACC-form." The circuit loads $a-1$ into the constant register rK. Then, the MACC is executed on each element of the X vector, and stored back into the array.

[4]With a 4096-byte page, 512 8-byte doubles can be stored. 3 are already used, leaving $509 \approx 254 \times 2$.

While using a MACC to do a simple multiplication is not the most efficient, the idea of this project was to implement as much as possible with homogeneous hardware. This ensures that the implementation of more complex functions such as Level 3 BLAS remains straightforward, and hardware routing does not become as much of an issue: new functions can simply use the nearest available MACC on chip, rather than having to route to a sub-functional unit of the correct type. Similar to the 90/10 statistic mentioned above for software execution, Marrakchi points out that, in FPGAs, 90% of chip area is dedicated to routing while the actual chip logic fits in the remaining 10%. This project did not address chip-level implementation, but it nonetheless makes sense to preemptively develop hardware that would make routing less of an issue in the future.

3.2.2 axpy

The Swiss army knife of BLAS, `daxpy` performs the operation $Y := aX + Y$. For compatibility with other functions, which assume X is the modified variable, `daxpy` was changed internally to $X := aY + X$. However, the arguments of the function call were also swapped by the compiler, ensuring functionally identical behavior.

Here, rK was set to a. Then, the MACC was executed on each element pair in X and Y, and stored back to the array.

3.2.3 nrm2

`dnrm2` returns the L2-norm (vector magnitude) of the X vector. This operation is performed in two parts: first, the sum of squares of the elements; second, the square root of that value. The former is a relatively simple MACC operation with X passed as both the second and third argument. Thus, each stage computes $rS := rS + rX \times rX$, with rS initialized to zero by default.

Computing the square root in hardware is slightly trickier. There are a number of methods for approximating square roots, but the Babylonian method (a special case of Newton's method) is particularly well-suited for MACC units. Starting with some estimate x_0 for the square root of S,

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right)$$

This can be repeated until a sufficiently precise result is attained. For bad estimates (i.e. using the value itself), the number of iterations to get to a certain precision is proportional to the logarithm of the value. Interestingly, however, by using a better initial estimate, maximal precision can be obtained in a constant number of steps (in this project, only five iterations sufficed).[5,6] Finding this estimate is straightforward in hardware thanks to the floating-point representation. Recall that floating-point numbers are stored in three parts: sign, mantissa M, and exponent E, such that

$$x = \pm M \times 2^E$$

where M is a fractional representation bounded by 0.5 and 1. We can estimate $\sqrt{x}$ by dividing E by 2 (right bitshift); this value will be at least $0.707... = \sqrt{0.5}$ times the actual square root.

This estimate is then fed to the MACC unit. Two extraneous operations (reciprocal and half) are required per iteration, but otherwise, the square root is computed very efficiently. See Figure 4 for the full algorithm.

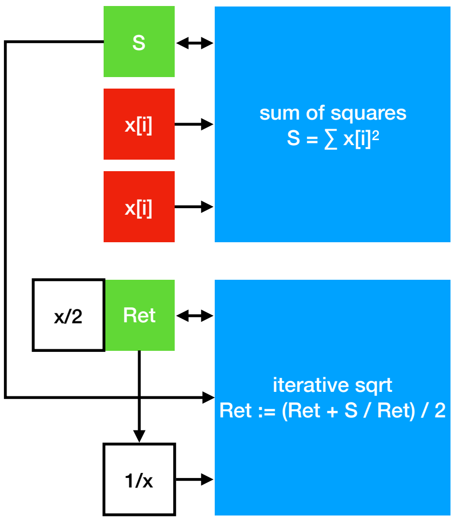

Figure 3: Block-level implementation of `dnrm2`

Latency estimates for `dnrm2` include these additional operations: 2 cycles for the approximate square root; 2 cycles for each reciprocal, and 1 cycle for each divide by two.

3.2.4 asum

This function sums the absolute values of the elements of X (this happens to be the L1-norm). Since no multiplication needs to occur in this implementation, `dasum` sets the last argument to the MACC to 1. And again, due to the floating-point representation, the absolute value does not needed to be computed: instead, the sign bit for the input register, rX, can simply be held at 0

[5]To test this, I ran a few hundred tests on a large range of values (from e^{-200} to e^{200}), and confirmed optimal precision was reached for every value.

[6]To be clear, this is a constant number of iterations for a specific bit length. The algorithm outlined above is technically linear in the number of bits in the input, but not the logarithm of the number. All `doubles`, for example, will have constant execution time.

```
// Executing rRet := rRet + rS * rR
Macc M_sqrt(&rRet, &rS, &rR);
...
// compute estimate M * 2^(E/2)
int exp;
double m = frexp(rS.get(), & exp);
rRet.set(ldexp(m, exp >> 1)); // div 2

for (int i = 0; i < SQRT_ITERS; i++) {
  rR.set(1 / rRet.get());
  M_sqrt.execute();
  rRet.set(rRet.get() / 2);
}

double r = rRet.get();
```

Figure 4: C++ implementation of MACC-based hardware square root. rS contains the sum of squares.

(signifying a positive number).

3.2.5 Other functions

The above five functions do not form the entirety of Level 1 BLAS, but they proved to be the most relevant for MACC-hardware implementation. Others, such as `idamax` and the rotation functions, were not conducive to MACC implementations, while others like `swap` and `copy` would see no benefit from optimized hardware implementations, since their software and memory overheads would not change.

4 LLVM Infrastructure

The LLVM Compiler system is an ideal framework for this project. Its modular design means that different stages of the compiler pipeline are decoupled, so modifications can be made at one stage without affecting others. This flexibility is largely facilitated by the LLVM Intermediate Representation (IR), a platform-independent language that acts as a convenient intermediary between high-level languages and assembly. Visually and functionally, IR is very similar to assembly, but has the benefit of being cross-platform and very human-readable.

In an LLVM-backed compiler, a *front-end* converts the source language to IR. A series of *passes* analyze or transform the IR, and a final *back-end* converts the IR to machine code. While this schema may seem simple, it is incredibly powerful. Separate compilers do not need to be written for every source language/target system pair (i.e. Python for Mac, Python for Windows, GCC for Ubuntu, GCC for Fedora, ...). Instead, any front-end is connected to any back-end, creating a full compiler. Cross-compiling (where the compiler creates code for a target architecture other than its native one) is also much easier, since the compiler itself is no longer reliant on the host machine.

Furthermore, LLVM IR can be interpreted by a JIT, which means that IR itself can function as a kind of platform-independent executable. Apple, a major benefactor of LLVM, uses this technology on the App Store: developers provide platform-independent IR, or something similar; when a user downloads an app, Apple creates a device-specific optimized binary.[7]

To familiarize myself with the LLVM ecosystem, I wrote a front-end compiler for a simple functional programming language called Kaleidoscope, following the introductory LLVM Tutorial.[8] I initially had intended to extend the Kaleidoscope language to a point where I could use it for my research. However, precisely due to the modularity outlined above, there ended up being no benefit to working with a (partially-implemented) custom language, as opposed to a mainstream existing one like C.

After the front-end produces a candidate IR for the program, it is passed through each of a series of passes; these are where traditional compiler optimizations are performed. Included with LLVM, for example, are such transformation passes as dead code elimination, constant propagation, and loop unrolling. But it is trivial to write custom passes; this is one of the major strengths of LLVM as a research tool.

The bulk of the software-end of DBTOFU is in the form of a Function Transform Pass; LLVM calls this pass for every function definition it sees. When using a JIT, the pass would be called each time a new function is added to the JIT (and only then).

The pass first finds the `main` function, and inserts a call to the memory-mapping function. Then, for each recognized function (`cblas_ddot`, `cblas_daxpy`, etc.), the pass removes the function call and replaces it with a series of loads and stores to access the memory-mapped functional unit.

Currently, DBTOFU performs simple string matching on function calls. But one of the most promising avenues for further development is in more advanced matching techniques. For example, the FU for `dasum` will only be substituted

[7]`https://help.apple.com/xcode/mac/current/#/devbbdc5ce4f`
[8]`https://llvm.org/docs/tutorial/index.html`

if a function by the name of `cblas_dasum` is found. This is not ideal – it is implementation-dependent and not robust to, say, changes in function calls between API versions.

A better scheme would use program semantics to detect substitution sites. In the above example, DBTOFU would look not only for `dasum` function calls, but also loops that iterate over arrays and and sum the absolute values of their elements. Arbitrarily complex extensions of this idea are possible. Thus, even inline (or unknowing but functionally equivalent) uses of the function could be optimized into hardware execution.

One of the major bottlenecks in this system is data transfer. To perform a vector-vector operation such as `daxpy`, both vectors must be copied to the memory-mapped IO space from their location within the program's memory. Then they must be stored into FU-internal registers, and loaded back after execution; finally, the vectors are restored from MMIO to their original location.

The first and last step of this process could be skipped by using shared memory, instead passing the FU a *pointer* to each vector, rather than the entire vector itself. With the current setup, this is not possible, since the hardware components have access only to physical memory addresses. But pointers to vectors would be virtual addresses, so another component, connected *before* the translation lookaside buffer (TLB), could provide an avenue for hardware access to local program data.

5 Other Software

5.1 JIT

The software description detailed above uses a static compiler rather than a JIT. Due to the simulation limitations, DBTOFU had to be run with statically-compiled programs, but the same process would apply. However, JITs can run nearly as fast as compiled binaries, and since overhead for DBTOFU only occurs when functions are defined, not when they are called, this system does not itself add significant execution overhead. At least in theory, then, DBTOFU could act as a run-time optimizer and translator, realizing the hardware-independent goals of this project.

The call

```
...
%dot = call double @cblas_ddot(i32 100, double* %uX, i32 1, double*
     %uY, i32 1)
...
```

is transformed into:

```
...
%map = load i8*, i8** @_io_map
%map_d = bitcast i8* %map to double*
%N = getelementptr inbounds double, double* %map_d, i64 2
store double 1.000000e+02, double* %N
%rX = getelementptr inbounds double, double* %map_d, i32 3
%rX_i = bitcast double* %rX to i8*
%uX_i = bitcast double* %uX to i8*
call void @llvm.memcpy.p0i8.p0i8.i32(i8* %rX_i, i8* %uX_i, i32 mul
    (i32 ptrtoint (double* getelementptr (double, double* null, i32
     1) to i32), i32 100), i1 false)
%rY = getelementptr inbounds double, double* %map_d, i32 103
%rY_i = bitcast double* %rY to i8*
%uY_i = bitcast double* %uY to i8*
call void @llvm.memcpy.p0i8.p0i8.i32(i8* %rY_i, i8* %uY_i, i32 mul
    (i32 ptrtoint (double* getelementptr (double, double* null, i32
     1) to i32), i32 100), i1 false)
%func = getelementptr inbounds double, double* %map_d, i64 0
store double 2.000000e+00, double* %func
%alpha_idx = getelementptr inbounds double, double* %map_d, i64 1
%dot = load double, double* %alpha_idx
...
```

Figure 5: Demonstration of BLAS call replacement. Values (`%...`) have been given more descriptive names for legibility here; internally, they are just assigned sequential numeric identifiers.

5.2 Memory-Mapped IO (MMIO)

The Linux system call `mmap(2)` maps files into memory as arrays. This allows positions within files to be accessed as array elements. However, by calling `mmap()` on `/dev/mem` – the memory pseudofile – the operating system will provide a direct map to a specified range of physical memory addresses. This is necessary for this project since the FU hardware has no straightforward way of accessing the calling process's virtual memory.

However, this mapping cannot occur at will. Directly accessing physical memory can be dangerous since other processes might have virtual memory residing at a conflicting physical address. Thus, the BIOS (in this case, a gem5 configuration file) is instructed to reserve a range of addresses for IO access. No virtual memory will ever be backed by that physical memory.

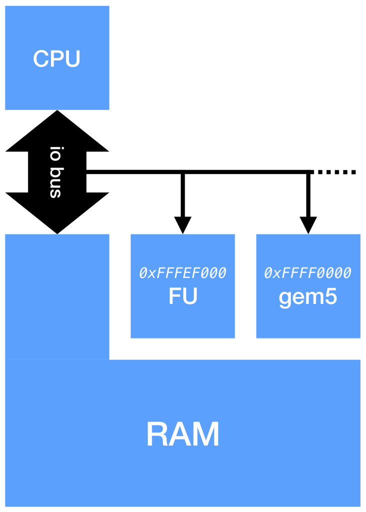

Figure 6: Modified memory topology

The memory bus is also modified, so that memory requests within a specified range are routed to the FU, rather than to the rest of the memory system. DBTOFU reserves one page of memory, from `0xFFFEF000` to `0xFFFEFFFF`, for communication with the hardware functional unit. Thus, all requests for memory at those physical addresses will be forwarded to the FU, rather than the memory system itself.

The first thing a DBTOFU-optimized program will do upon executing is to create a local map to that IO page, returned as an array. Each BLAS function can then read from and write to array elements to communicate with the FU.

5.3 Firmware

Minimal device-level firmware is required for DPTOFU's operation. The page reserved for memory-mapped IO must be reserved by the BIOS, to prevent any other accesses to those physical memory addresses. In gem5, this is a matter of modifying a configuration file, but in a physical system, a BIOS software update would be required.

In a commercial implementation of DBTOFU, some minimal firmware would communicate the available functional units to the JIT or compiler. Currently, the few functions I chose to implement were hard-coded into the compiler. Ideally, the CPU would identify its capabilities to the operating system, which could in turn forward this information to the compiler by way of a system call or pseudo-file system. DBTOFU would need to know the names (or other matching criteria) or relevant functions as well as their function signatures.

While not directly related to this project, I wrote some minor additional firmware for the simulated CPU. The implementation of BLAS used takes advantage of the x86 `fsqrt` (floating-point square root) instruction. While this instruction exists on real chips, it was not included in the gem5 instruction set, so had to be implemented. This is not very complicated; gem5 implements instructions as Python code, so only some minimal decoding and register-mapping logic had to be added.

5.4 Git

All code for this project is available online via the Yale Git server: `https://git.yale.edu/emb99/eecs-senior-project`. Also included in this repository is a file, `notes.md`, which contains my full notes on this project, on everything from background research, to system setup, to testing.

6 Results

Unfortunately, DBTOFU does not perform any better than the "unoptimized" version, performing as much as 10% slower, but occasionally 5% faster, in some test cases. This is also true across many other metrics, such as execution time, memory usage, memory bandwidth, and energy. But this should not be surprising: in its current iteration, DBTOFU takes advantage of very few optimizations that are available to it. It is instead an extensible framework for building more efficient systems that take advantage of parallelization, shared memory, and even more complex functions – for BLAS Level 1, memory overhead is significant and computational workloads are minimal.

The first step in making DBTOFU more efficient would be to redesign the memory system. The current implementation can take an order of magnitude longer to transfer data to the FU than to execute the function. This example

N	Normal prog.	DBTOFU	ratio (DBTOFU/norm)
1	14M	15M	1.045
2	16M	16M	0.999
4	20M	20M	1.014
7	29M	29M	1.012
11	40M	41M	1.042
17	55M	56M	1.019
26	79M	81M	1.029
40	116M	118M	1.017
61	172M	174M	1.011
92	257M	258M	1.003
139	383M	359M	0.938
209	565M	535M	0.945

Table 1: Cycle count for Angle Test

result was measured with $N \approx 200$ for `dnrm2` and `dasum`; memory transfer consumed around 16,000 cycles while execution took only 1,600 cycles. Functions such as `daxpy` that must copy the entire array back would fare even worse; having a shared memory model would be invaluable.

Representative results for two test programs, in Tables 1 and 2 are included. The former computes the angle between 100 pairs of randomly-generated angles (this uses `ddot` and `dnrm2`). The latter performs various operations on 100 pairs of vectors, and then normalizes their magnitudes (using `daxpy`, `dnrm2`, and `dscal`). Similar results were seen for other metrics: DBTOFU usually performs slightly worse than baseline for small N, but for larger vectors, begins to outperform the original program. The number of function calls also did not improve dramatically, as expected (DBTOFU, ideally, should have saved significant time by not using any function calls), likely due to the necessity of `memcpy`ing vectors.

I suspect there is some threshold crossed around $N \approx 100$ where the costs of software computation begin to outweigh the cost of memory transfers, and the benefits of hardware computation win out. Unfortunately, larger vectors did not fit in the current memory-mapped IO layout, and would need to be tested with a shared-memory system instead.

N	Normal prog.	DBTOFU	ratio (DBTOFU/norm)
1	15M	15M	1.025
2	14M	15M	1.060
4	19M	20M	1.035
7	28M	30M	1.091
11	38M	42M	1.105
17	54M	60M	1.106
26	78M	86M	1.094
40	115M	125M	1.084
61	171M	186M	1.087
92	259M	275M	1.059
139	378M	367M	0.968
209	566M	546M	0.964

Table 2: Cycle count for Vector Test

7 Further Work

One of the main premises of this research is that significant developments in computer architecture are hindered by recompilation. The LLVM pass described above, therefore, is viable only if it is run within a JIT. Unfortunately, I was not able to run DBTOFU in a JIT on the simulated system.

The issue here came down to often-dreaded dependency issues. To run the LLVM native JIT, `lli`, on the simulated system required that system to have the appropriate dynamic libraries available (matching the host machine, where `lli` was compiled). Recompiling those libraries, or compiling LLVM in its entirety, on the simulated system, was infeasible for a variety of reasons (such as speed, disk usage, and language dependencies). I made some attempt to install LLVM directly onto the simulation disk image, but ran into kernel- and filesystem-version incompatibilities. The final possibility was to compile `lli` statically, thus removing the dynamic library dependency. But this attempt, too, failed, since statically-compiled JITs do not work under the current LLVM release. Whether this is a bug in the symbol loader, or a fundamental inconsistency in JIT architecture, it became infeasible to pursue, so I instead used statically-compiled binaries, with the DBTOFU substitutions applied at compile time. With some more work, and particularly more knowledge about JITs and static compilation, `lli` should be able to run under simulation; then, more realistic data can be collected on DBTOFU.

DBTOFU currently does not take advantage of parallelization; all hardware implementations run in $O(N)$ time. This is a ripe avenue for further research, since hardware circuits are not limited by the normal synchronous pipelines of computers. Many of the BLAS functions are particularly well-suited for parallelization, as well. `daxpy` and `dscal` have no inter-element dependencies, so work could be divided evenly and quickly among many parallel MACC units.

Many memory improvements are possible, as mentioned above. Among these are the use of shared memory to remove the significant copy bottleneck, the introduction of a separate cache for DBTOFU, to speed up repeated accesses (for example, if multiple subsequent operations use the same vector), and extended cache accesses, to allow the unit to access entire vectors, rather than individual cache words (which would likely be much shorter than a vector).

Finally, DBTOFU's function as a dynamic binary translator could be immensely expanded with smarter matching. Currently, only function-name string matching is in place, which limits the scope of the system. A more advance system should recognize program semantics, and replace entire blocks of code for which functional hardware equivalents exist.

8 Conclusion

Even though DBTOFU did not succeed in outperforming the software implementations of BLAS, all is not lost. The fact that a largely unoptimized hardware system could match pace with the highly-optimized OpenBLAS implementation is encouraging. With even some of the minimal improvements outlined above, DBTOFU could easily meet significant performance benchmarks.

DBTOFU may see some use in an entirely different field: embedded systems. Instead of trying to optimize math functions, this system could be used as an intermediate compiler for devices that already take advantage of the kind of memory-mapped registers that were used here. Currently, those devices rely on a device-dependent preprocessor macros to access this functionality from C. But technology similar to DBTOFU could instead recognize a standard function library, with the specific hardware implementations provided by the chip itself.

Finally, it is important to also recognize the limitations of simulators such as gem5. Nowatski, et al. emphasize this point in their paper "gem5 [...] Considered Harmful"; while they focus on specific issues that may not be relevant to this research, their general arguments about taking simulator results with a grain of salt are nonetheless valid. For a system as intricate as an x86 processor – especially a *modified* processor – simulations are useful but can never be as accurate as the real thing. I am relatively confident in my latency estimates, but as DBTOFU itself becomes more complex, possibly integrating more into the memory system, and exploiting parallelization, such first-order timing estimates will become less useful. Future versions of DBTOFU should eventually be run on an FPGA, or, if possible, in silicon.

9 References

"Kaleidoscope: Implementing a Language with LLVM." https://llvm.org/docs/tutorial/index.html

"LLVM Language Reference." http://llvm.org/docs/LangRef.html

Lowe-Power, Jason. "gem5 Tutorial." http://learning.gem5.org/book/index.html

Marrakchi, et al. "FPGA Interconnect Topologies Exploration", International Journal of Reconfigurable Computing (2009). http://dx.doi.org/10.1155/2009/259837

Mei, et al. "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix," Lecture Notes in Computer Science. https://doi.org/10.1007/978-3-540-45234-8_7

Nowatzki, et al. "gem5, GPGPUSim, McPAT, GPUWattch, 'Your favorite simulator here' Considered Harmful." https://research.cs.wisc.edu/vertical/papers/2014/wddd-sim-harmful.pdf

"The Often Misunderstood GEP Instruction."
http://llvm.org/docs/GetElementPtr.html

Sampson, Adrian. "LLVM for Grad Students." https://www.cs.cornell.edu/~asampson/blog/llvm.html

10 Acknowledgements

I am indebted to my patient advisor, Prof. Rajit Manohar, and the DUS of Electrical Engineering, Prof. Mark Reed. Rui Li of Prof. Manohar's AVLSI Group generously provided many hours of guidance on gem5 and LLVM.

The inspiration for the MMIO system came from gem5 itself, which uses a similar mechanism for communication between host and simulated machines.

I would also like to thank Lang Hames, Adrian Sampson, and the users of Stack Overflow for their crucial assistance navigating the dark corners of LLVM.