Programming Reversible Computers

Ph.D. Thesis Proposal

FIRST DRAFT
$Revision: 1.39 .

Michael P. Frank
MIT AI Lab
545 Technology Sq.
Cambridge, MA 02139

Started August 27, 1996.
Formatted on December 5, 1996.
Last RCS checkpoint prior
to formatting this version was on
$Date: 1996/12/03 16:44:14 .

A current version is available online at
`http://www.ai.mit.edu/~mpf/proposal/proposal.html`

[The following "abstract" is less concise than might be desired. If I move the contents out of here, though, I would like to include them somewhere else, perhaps as the first part of the Introduction.]

Abstract

I propose a program of Ph.D. thesis research on the theory and practice of how to easily and effectively program computers that are either fully or partially reversible, *i.e.*, that either discard no information during their operation, or that do so only occasionally, and in a carefully controlled way.

The main reason that reversible computers are interesting is that the fundamental physics of computation teaches us that an increasing degree of reversibility will be essential in future computing technologies, as they approach the ultimate limits on computational density[1] given by the laws of physics (as they are currently understood). As our computing technologies approach these limits, and reversibility becomes more ubiquitous, we will need programming paradigms that can flexibly accommodate ever-increasing degrees of reversibility, up to and including full reversibility. Although those limits remain distant, enough is understood about them already to begin developing appropriate programming frameworks now, certainly with regards to the management of reversibility, at least. So that is the primary long-term motivation for this research.

But full reversibility may have shorter-term applications as well, for needs such as error detection, auditability, and recoverability, either in distributed systems utilizing optimistic concurrency, or in multiprocessor CPUs using speculative execution, or in databases supporting atomic transaction processing. (All of these contexts really amount to the same thing.) The reversible programming framework developed in this research will aim to be particularly useful for these needs, independent of the longer-term applications in physically-limited computation.

The research proposed here will take place as part of the MIT Reversible Computing Project,[2] which is mandated to (1) demonstrate a fully-reversible, energy-recovering CPU in silicon, (2) investigate the applications of reversible computers in the areas of low-power and verifiable computation, and (3) explore the connection between reversibility and the fundamental physics of computation.

The core of the work proposed here will be to design and implement appropriate programming abstractions, compilation techniques, and run-time environments for facilitating efficient computation with arbitrary degrees of reversibility. Test programs will be written within the framework,

[1] "Computational density" meaning, perhaps, the number of useful computational operations possible per unit space, and also per unit time, and also per unit mass/energy included in the system.

[2] The MIT Reversible Computing Project is sponsored by DARPA contract DABT63-95-C-0130. For more information about the project, see `http://www.ai.mit.edu/~mpf/rc/home.html/`.

to demonstrate, for example, how to efficiently simulate dynamical physical systems in a fully-reversible fashion. Other test programs may demonstrate how reversibility can be used to facilitate concurrent transactional operations. I have already worked on the design of the programming framework and the test programs to some degree; in the body of this proposal I will present some of the work performed so far.

The test programs will then be executed, of course, first on simulated reversible processors, and then later on the real reversible processors that are being built as part of the MIT Reversible Computing Project. The test programs will serve to exercise those processors, as well as the programming framework. Data will be collected allowing us to quantitatively compare the efficiency and power dissipation of our reversible programming framework to that of irreversible computers and less sophisticated approaches to reversibility.

The thesis document itself will motivate and describe the design of the programming framework, discuss the sample programs in detail, report on the results of the testing, and mathematically analyze the various tradeoffs (such as space vs. run-time efficiency) accounted for in the design. It will also discuss the relevant physics of computation, and, if there is time, discuss the impact of real physical limits on the theory of computational complexity. The conclusion will point to directions for future work. For example, the current project is based on a serial, von-Neumann-style architecture; but we would ultimately like to have massively-parallel reversible computational architectures, that could take better advantage of the sort of uniform, parallel, locally-interconnected, cellular-automaton-like computational substrate that is thought to be the fundamental computational resource that is provided to us by the laws of physics.

Contents

[The proposal as a whole needs more references, more details on my current thoughts, and I'm sure a whole lot of miscellaneous reworking and clean-up. But at least now all the sections have been filled in to some degree, and the proposal can now be considered "complete" (if rough-edged), for purposes of a first draft. The next step is to get feedback from the readers and then do major rewrites on the basis of that feedback.]

1 Introduction & Background

As the intended readers of this proposal already well know, a *logically reversible* computational process is one that does not discard any information as it proceeds, so that its prior state is always uniquely determined by its current state. (In the thesis I will formalize this notion and state it much more precisely, but let's not worry about that at the moment.)

1.1 History of Reversible Computing

The study of logically reversible processes has historically been motivated by concerns in physics; for example, the proper resolution of the famous "Maxwell's Demon" paradox of thermodynamics required understanding that a system's method for disposing of unwanted information is an issue that can not in general be ignored when considering the thermodynamics of the system.

Detailed accounts of the Maxwell's Demon paradox, along with many other historical developments in the study of reversibility and its relationship to physics, can be found in the literature (see for example [4] and references therein), and so I will refrain from recounting those developments here in the proposal. The actual thesis will, however, review the history of the field in whatever degree of detail the readers feel is desirable.

Instead, for now, let me just briefly summarize the state of our current knowledge, as I understand it, about reversible computation and its relationship to physics.

1.2 Physics and Reversible Computation

Physics is reversible. Physics itself, when considered at the micro-level of the dynamics of individual particles, rather than at the macro-level of thermodynamics' statistical ensembles, appears, so far as we know today, to be totally reversible.[3]

Therefore, the evolution of the state of the universe does not itself lose any information; so when we say a piece of information has been "discarded," we must not mean it has been lost at the micro-level, compressing the state-space, but rather something else. Usually what is meant instead is that the information has been transferred.

[3] Some extrapolations of known physics claim that information can be irreversibly destroyed by black holes, but to my knowledge, the evidence in support of these extrapolations is weak.

Originally, the information of interest resided within the controlled, known, observable part of the universe's state that constitutes the state of our computational system of interest. Later, the information has moved, outwards to some uncontrolled, unknown part of the universe's state, such as to the thermal degrees of freedom of the particles in the computer or its environment. The "discarded" information has really just become all mixed up with the unknown information that was already present in the environment, in typical chaotic fashion, to the point where it appears random, and we can no longer effectively recover the original information at all.

Increasing the information in thermal motion implies increasing the energy of that motion, and so in addition to getting more mixed-up, the non-computational components of state become more energetic as well. This random-looking, micro-state information and energy is what we call "heat."[4]

Therefore, when information is discarded by a computer in the usual fashion, the result is the production of heat. In the standard analysis, a minimum amount of energy $E = k_B T \ln 2$, where k_B is Boltzmann's constant and T is absolute temperature, is required to encode 1 bit of information, and this energy, together with the information it carries, moves from being ordered energy in the computer to becoming heat in the environment.[5]

Actually, current computers typically use around six orders of magnitude more energy than this theoretical minimum to store a bit of information in one of their circuit nodes, and they throw away all of this energy each time the stored value is changed.

Computation can be reversible. Fortunately, it turns out that the discarding of information (and the energy used to store it) is not actually required for computation. On a theoretical level, the simple expedient of saving a history trace of past states suffices to transform arbitrary Turing machines [3], cellular automata [12], or recursive function systems [cite PhysComp lambda calculus paper] into equivalent reversible forms.

Saving this history information generally leads to increased storage requirements over time. However Bennett[5] has shown that the rate of increase can be kept low, at the cost of decreased computation speed. Moreover, as we will discuss later, there are interesting computations,

[4] I'm still pretty unclear on exactly *why* greater thermal kinetic energy implies that the particles involved are carrying more information. Digitally speaking, sure, if a particle is moving faster, it takes more bits to express its velocity to some fixed absolute precision. But if velocities can be real numbers, then it takes infinite information to express them no matter how small they are. So why should there be fewer low-energy states than high-energy ones (and thus less information in them)? I suppose that if the state space is made discrete and finite this all goes away. But how is this paradox dealt with in standard, continuous physics with an uncountable state space?

[5] I'm actually still hazy on the reasons for this too. Need to ask readers some questions about this.

such as physical simulations, that can be performed reversibly and run indefinitely without incurring any increase in storage requirements at all.

1.3 Computation in the Limit

The discoveries mentioned above raise the question of whether computer technology might be enabled to approach its ultimate physical limits more closely by operating reversibly, and refraining from intentionally transferring the energy associated with its state information from its computational degrees of freedom to its thermal degrees of freedom. (Let's say "modes" rather than "degrees of freedom" hereafter.)

Transferring energy to heat in a computer is generally undesirable because unless the heat can be removed quickly, the increased temperature will eventually interfere with the carefully-controlled parts of the computer's state that are performing the intended computation. As computer components become packed more densely, the removal of heat becomes more difficult.

Of course, some amount of unintentional coupling between computational and thermal modes will always occur in any system where the two are not well isolated. For example, in normal conductive but non-superconductive materials, a flow of charge will be strongly coupled to thermal modes in the conductive material, and so any computer that progresses through its state space by moving charge around along such conductors will dissipate energy, at a rate that increases with the rate of charge transfer.

In contrast, fast low-dissipation computers would need to be primarily *ballistic*, meaning that the energy of the computer's motion through its computational state-space would not be strongly coupled to thermal modes in the environment. Yet, even a ballistic computer will have to dissipate $k_B T \ln 2$ per bit of information discarded. So to really push the minimum energy limits, a ballistic computer will need to be reversible to some degree.

Therefore, if we are interested in what the best computers will ultimately look like, in the long term, we are faced essentially with a spectrum of possibilities:

- Non-ballistic, non-reversible, highly dissipative computers requiring extreme rates of heat removal.
- Ballistic, reversible, and almost totally non-dissipative computers, needing very little heat removal.
- Something in between (mixing elements from each).

An interesting question is to try to ascertain, given our knowledge of physics at this time, which of these possibilities is ultimately going to be the most useful. Certainly, the first approach is the most useful at present, given current technology. But in terms of truly maximizing the amount of useful computation performed per unit time given a finite supply of mass-energy and available space, it seems fundamentally inefficient. In dissipating unwanted information to the form of heat internally, the possibility is opened that the unwanted thermal energy will interfere with normal computational operations. If conductive heat removal through the normal components of the computer is not sufficient for the task, then additional space, materials, and energy in the computer will need to be devoted to some form of active heat removal (for example, circulating coolant through channels in the computer and through an external radiator).

The ballistic, reversible computer, in contrast, produces little heat, but still lots of garbage information. This information is generally still useless to us; whether it's in the form of heat, or in a more controlled form, and must still be gotten rid of somehow so that it does not eventually use up all the available degrees of freedom in the computer. Simply emitting all this information from the system as it is produced in its raw, original form will present essentially the same kinds of demands for extra resources that would be required if the information being emitted were in the form of heat; for example, some form of conduit would still need to be devoted to its removal.

However, the situation is improved in several ways if the garbage information is being stored in degrees of freedom that can be observed and manipulated, rather than being mixed up with the unknown state of thermal modes in the environment. For one thing, the removal of the unwanted information might occur in a faster and more energy and space-efficient way than if it the unwanted information is in the form of heat. For example, energy storing the unwanted information could be emitted from the system (for purposes of either external storage or dissipation) in a ballistic fashion, probably with less requirements for additional energy, mass, and volume than in the case of circulating coolant.

Even more importantly, the reversible computer opens up the possibility of vastly reducing the amount of extra information that actually needs to be removed from the system. Current computers tend to discard information in nearly every individual logic operation. But the reversible computer can accumulate garbage for a while, copy a higher-level intermediate result, and then run the previous sequence of operations in reverse to reversibly clear the garbage, leaving only a higher-level intermediate result to deal with. This process can be repeated at higher levels, although at a computational cost that increases polynomially in the garbage-compression factor that is achieved.[6] But the quantity of garbage may also be usefully reduced via other means, such as traditional data compression. Moreover, special kinds of computations, such as simulations of reversible physical systems, may be performed with no accumulation of garbage whatsoever, as I will demonstrate in the thesis work. (See §5.3.1.)

[6] I'd like to write out a proof of this assertion in the thesis if I have time. It would be a corollary to the existing analyses of Bennett's garbage reduction algorithm.

1.4 Practical Implementation Technologies

Although we are still a long way from being able to build fast ballistic reversible computers that approach the limits of what is possible given the laws of physics, we can build today a non-ballistic reversible computer in a current electronic technology, using the techniques of adiabatic circuits, which nicely demonstrate how to eliminate that part of the energy dissipation that is due to the discarding of information.

I could, at this point, survey the literature on adiabatic circuits, and I should probably do so in the actual thesis, but for the purposes of this proposal, I will just say that adiabatic circuit techniques are rapidly developing, and are becoming more practical.

1.5 The MIT Reversible Computing Project

The theoretical importance of reversible computing in the long term, together with the advent of practical reversible circuit techniques, have spurred the creation of (what I call) the MIT Reversible Computing Project. This project aims to use a pipelined adiabatic circuit technique that was developed earlier in our group [14] as the basis for designing and fabricating a fully-reversible, 32-bit microprocessor, called Pendulum [13], to provide a convincing proof-of-concept to demonstrate that designing and implementing a fully-reversible architecture is not unreasonably difficult.

The project will also work with other researchers on the subject of the physics of computation, characterize the low-energy benefits of reversibility, and investigate the possible applications of reversible computing for trustable/verifiable computation. My own thesis work, will focus on the more software-oriented later items, and on how to effectively program reversible computers in general, although I have also been involved with the design of the Pendulum instruction set and in building prototype implementations.

For more information about the project, see the web pages at the URLs `http://www.ai.mit.edu/~mpf/rc/home.html` and `http://www.ai.mit.edu/~cvieri/reversible.html`.

That concludes our general introduction and discussion of background. Let us now proceed with the main body of the proposal.

2 Summary of Proposal

The remainder of this document proposes my specific program of research, which is designed to fit within the overall aims of the MIT Reversible Computing Project, and yet to also constitute an independent line of original research, distinct and separable from the work performed by other members of the project. In this section we summarize the proposed research.

2.1 Motivations & Applications

One of the ongoing aims of the proposed research will be to seek out and elucidate some convincing (or at least plausible) near-term motivations for the study for reversible computing techniques in general, and for fully-reversible computing in particular. So far, the motivations have more to do with the ultimate physics of computing, and general scientific inquisitiveness, rather than with immediate practical needs. However, we feel that there are several real-world needs for which reversibility either may be useful already, or will become useful after a while, as requirements (e.g. for lower-power operation) become more and more stringent.

If the search for concrete near-term applications is successful, then the first major chapter of the thesis (after introduction and background) will present the analysis that shows why reversibility is useful for those applications. This analysis will help to motivate the development of the later details. If the search is, however, not successful, then the analysis of the reasons why reversibility may not be so useful in practice in the near term will be moved towards the end of the thesis, to a results/conclusions/future work section, and the opening of the thesis will emphasize the general scientific justifications for the study of reversible computing from the point of view of studying the physics of computation.

Section 3 of this proposal will outline the current state of my ideas regarding particular near-term motivations for reversible computing research, items such as power savings, error detection, and immunity from intentional destruction of data.

2.2 Instruction Sets for Reversible Computers

Once the motivation for the research has been presented, the thesis body will then proceed to discuss reversible computer programming issues, in a logical progression from the lowest to the highest levels. The first chapter will discuss the design of instruction set architectures for reversible computers.

The chapter will first review the known existing instruction sets, in particular the Pendulum instruction set invented by Carlin Vieri, which will be the instruction set implemented in the MIT Reversible Computing Project's processor design work. I will point out interesting features of the Pendulum instruction set, one or two of which were suggested by myself to support particular programming needs.

Next, the chapter will discuss alternative forms of reversible instruction sets, and new features that might be useful but are not currently supported by Pendulum. These include: (1) software control of hardware execution direction; (2) special hardware I/O instructions (which the implementation is encouraged to support reversibly, when possible); (3) an explicit, optional "garbage disposal" instruction, which the hardware is encouraged to support using as little resources as possible, and which breaks program-level reversibility, but which would allow another dimension of flexibility for power-saving applications in which actually reversing over the program is not required; (4) an instruction set guarantee of full reversibility regardless of program correctness, which could serve as a safety net in case the OS and/or compiler software is not deemed completely trustworthy.

The work on higher-level programming interfaces, discussed later in the thesis, will attempt to take into account a wide range of possible options for reversible instruction sets, and will support whatever changes in the Pendulum instruction set might be necessitated by implementation considerations. The compiler will operate in a modular, instruction-set-independent fashion, in which different reversible ISAs can be plugged in easily, as desired.

2.3 High-Level Languages for Reversible Computers

[This is getting too big for a summary. Move it to main body and write a shorter summary here?]

After the various possibilities for reversible instruction sets have been discussed, the thesis will cover higher-level programming issues. These will include (1) giving the high-level programmer the flexibility to explicitly reverse and reclaim storage himself when required, (2) giving the programmer the ability to use irreversible abstractions when desired, with a reversible implementation ensured by the compiler and run-time system, (3) allowing full reversibility to (optionally) be maintained using only logarithmically-increasing space and small-polynomially-increasing slowness, using Bennett's algorithm.

A example of a high-level reversible language that provides all of the above features will be designed and implemented. The design will be discussed. To show the suitability of the language, a number of sample programs will be written.

One sample program will show the usefulness of giving the high-level programmer the ability to access the machine's reversibility, by showing how one could use this ability to program an indefinitely-continuing, totally reversible simulation of a physical system without leading to ever-larger storage requirements or ever-slower run-time performance.

Another version of the same simulation will be written in a way that forces the compiler and run-time system to manage all of the garbage data, and will thus demonstrate the inefficiencies that can result from hiding too much of the machine's underlying reversibility from the programmer.

Other sample programs, for applications in which the reversibility cannot be so easily maintained by the application programmer, will demonstrate the benefits of having some sort of limited hardware support for garbage disposal. We will simulate an architecture in which there is a hardware garbage-disposal instruction, and then for that architecture we will allow the compiler to throw away some data, using that instruction.

We hope to show that the compiler and run-time system can, via a combination of Bennett's techniques, standard data compression, and hardware garbage disposal, be made to drastically decrease the amount of garbage data that is dissipated, while still maintaining reasonable (not asymptotically slowing) run-time performance.

Parameters input to the compiler can specify a desired rate of garbage data production, and the compiler will compile the program so as to perform useful computational work as rapidly as possible while still not exceeding that garbage production rate. (I say compiler, but the actual capability for flexibility may reside in the run-time system.) We hope to show that, under some reasonable assumptions (such as refraining from detailed program analysis), our implementation yields optimal run-time efficiency in this regard, relative to the particular data-compression algorithm being used.

2.4 Operating Systems for Reversible Computers

[This is getting too big for a summary. Move it to main body and summarize further?]

The preceding chapter, on high-level programming languages, will have considered how to run a single program in isolation on a reversible computer, with a single stream of input and output. But this begs the question of how to do what is now standard on regular non-reversible computers: run many programs simultaneously, all talking to multiple input sources, and output devices, and to each other, sharing resources efficiently and not interfering with each other's internal operation.

It is unrealistic to hope that we can design a full-featured operating system in the time allotted for this thesis research. It also would not be very interesting to reinvent all of the obscure, grungy details of modern operating systems. However, there are a few very fundamental issues about how reversibility and operating system requirements interact, that we will hope to be able to address, at least on a conceptual level.

The obvious interesting problem comes up when multiple, intercommunicating reversible programs are running at once on one computer. Let us even assume that issues of local process address spaces and memory protection are already handled on the hardware level, and that there is a distinguished system mode capable of manipulating process state. Simple issues of managing asynchronous interrupts and process scheduling will probably not be a problem (although the issue ought to be thought about in more detail).

No, the real problem occurs when one of the processes wants to actually run in reverse. Reversing a code section that does no communication is not a problem. But imagine a program running for a long time in an environment where it is sending and receiving data from other processes. To reclaim storage, or check program results, the run-time system may wish to run a large section of a program's execution, perhaps even the entire program, in reverse. But how can that individual program be reversed when part of its execution has required interactions with other programs? Will those other programs need to be reversed as well? If so, should the reversing of other programs be done in an on-demand basis, or in synchrony? How can we maintain a consistent state across multiple programs in an efficient fashion?

We believe that to obtain the proper answers to these questions, we will need to look at the database and transaction processing literature, since there is a similar issue in that field of how to correctly roll back transactions in a database environment in which multiple concurrent transactions with the database are taking place, and when one transaction fails, its effects must be reversed in order to restore a consistent database state. In a reversible computer, this just becomes a special case of the more general problem of how to allow inter-process communication between reversible processes, which need not be database servers and clients in particular.

We hope that the study of the methods of transaction processing will lead to a useful synergy, in that once we know how to implement the methods of transaction processing at the lower, operating-system/IPC level that may be required for reversible multitasking, it will then be possible to go back and show how implement transaction-oriented databases in a simple way under our reversible operating system. One might implement rollback of transactions by just executing the code that performed the transaction in reverse, thereby (through the run-time system and operating system) reversing any interactions with other database processes appropriately as well. It is hoped that this technique may simplify and clarify the relationship between transaction processing and database design. However I am currently very new to the field, and will require a much closer look to be sure.

If we are successful in figuring out how to apply transaction-processing ideas to allow reversible multitasking to take place in a meaningful way, we will attempt to write a simple initial implementation for a multitasking run-time system on top of the reversible high-level programming language which will have been developed earlier in the thesis work. The implementation will ignore most of the unrelated OS issues such as asynchronous scheduling and the separation of user processes' address spaces, since these facilities are currently assumed to be straightforward to implement in a reversible way (we will need to justify this assumption in more detail in the thesis, however) and are therefore uninteresting from our perspective of solving more fundamental problems.

We might then build a simple transaction-processing database application on top of this reversible multitasking framework, in a way that uses OS task-reversal support to implement rollback. I am not yet clear on what might be an interesting example of a database application that would illustrate any advantages to the technique of providing transaction rollback via our OS support for general reversibility of tasks in a multitasking environment. However, if such an application does come to light, it would be nice to implement it as well, on top of this database facility, and demonstrate that it runs successfully.

Other examples of the use of the multitasking environment would be nice to have as well, but I am not yet clear on what an appropriate example might be.

2.5 Reversible Complexity Theory

[The following discussion needs to be rewritten in light of some recent discussions with Norm.]

An important open question for reversible computing theory is to more broadly characterize the range of possible computations that can be performed on reversible computers without incurring asymptotic overheads in time and space complexity, and to quantify the increase in space complexity that is incurred in other computational tasks (if any) in which some such increase is required in order for reversibility to be enforced.

If the results of such an extension of traditional computational complexity theory indicated that the computational complexity for some hard problems of interest (other than simulations of reversible physical systems, which we already know about) was not increased significantly when garbage data was taken into account, then that would mean that those problems could be solved on a fully-reversible computer with no heat production, without requiring significantly more computational resources than if the computation was performed in an irreversible fashion requiring extensive cooling. Therefore the discovery of such problems would further validate our premise of the importance of reversibility for computing.

At the moment I have not written a full section later in this proposal to convey my current thoughts on this topic, since I really haven't thought much about it. However, after pausing a moment just now to see if I could come up with any answers right away, I realized that the naive exponential search algorithm for solving a particular NP-complete search problem (namely SAT) can in fact be transformed into a reversible analogue that can solve the problem without any asymptotic increase in time or space complexity, as compared with the irreversible algorithm. (I will include the reversible version as one of the sample reversible algorithms I will write in my reversible programming language.) However, the extent to which this result can be generalized is unclear. It would be much more impressive if we could show that the asymptotically *best* known algorithm for the SAT problem could be rewritten reversibly without increasing the asymptotic complexity, or even better, if we could show that all search algorithms in some broadly-defined class had this desirable property.

Although this line of research is very interesting and important, I have not decided to definitely pursue it as a major part of the thesis. It seems to me that it would be difficult and time-consuming enough that it would force me to abandon many of the other thesis goals that are equally interesting, but less difficult. Perhaps it could more fruitfully constitute a topic for postdoctoral research.

2.6 Physics and Computation

Finally, we may also want the thesis to express any new insights about fundamental questions of computer science, physics, and the relationship between them, insights that we may stumble upon while thinking about how to program reversible computers.

For example, there is the interesting question of how the apparent reversibility of microphysics is compatible with the fact that macroscopically, we see a clear distinction between the forwards and backwards time directions. In the thesis, I may discuss my thoughts on how concept of "forwards time" and "increasing entropy" can be related to the increase in the formal informational complexity of the universe's state, as one moves away from the informationally simplest state along a bi-directional, reversible chain of states related by a computational transition function corresponding to the laws of physics. (More details in a later section of the proposal.)

[Anything else on physics?]

2.7 That's it for the Summary.

After discussing all of the above topics, the thesis will of course wrap up with a conclusion and suggestions for future work.

Now, all that was just a summary of the proposal; let us now move on to the main body of the thesis proposal, where we will get a bit more detailed. Each of the following sections will detail my current thoughts relating to the corresponding thesis chapters, as outlined above.

3 Applications of Reversible Computing

Before getting into the details of the design of programming paradigms for reversible computers, the thesis will first attempt to analyze the various plausible applications of reversibility, since the results of this analysis will naturally color and shape the the later work on how to program reversible computers in a way that can effectively address the aims of the different possible applications.

In reality, the studies on programming will also guide the discussion of possible applications, since discovery of what is feasible and what is not feasible at the programming level will suggest new applications and eliminate others. However, for expository purposes, we will discuss the applications first, since this will provide the motivation to hopefully entice the reader into looking closely at the later details.

Although the work is still in its early stages, we can already make a number of observations regarding the plausibility of various applications of full reversibility. I will present these now, so that feedback can be given to aid further development of these ideas.

3.1 Power Savings

One of the interesting potential applications of reversibility in computation is to reduce the energy dissipation per computational operation. [The Introduction will have already discussed how this works.]

There are two independent reasons why reducing energy dissipation might be a good thing: (1) preventing overheating, and (2) conserving a limited energy supply. Let us consider these separately.

3.1.1 Preventing Overheating

Heat production in modern microprocessors is becoming a serious concern. Many of today's high-performance processors internally dissipate tens of watts of power. They could easily dissipate much more, if not for the fact that designers are being forced to restrict the capabilities of their processors in order to prevent overheating. High temperatures would be anathema to the proper functioning of these processors. Heat-conductive and radiative packaging and air-cooling is needed to allow these processors to function.

Reversibility clearly helps with this situation, since any information-carrying energy that is conserved, or recovered adiabatically, is energy that does not go to heat. Fully reversible processor circuitry could, in principle, dissipate almost no heat, and could thus be packed as densely as fabrication techniques allow, without concern for overheating.

However, there are several points that may weaken the case for full reversibility for this application.

1. Full reversibility may be overkill. One important point is that the majority of heat dissipation in a processor is due to the charging and discharging of the many logic nodes internal to the processor. The programmer-visible information, such as is stored in registers, is small in comparison. So one could obtain a very large savings just by being reversible at a low level, say in instruction execution—but after the energy recovery from this process is finished and a new register value is obtained, the old state could be thrown away, dissipating a little heat, but not much in comparison to a full-blown irreversible-logic implementation.

The exact line between reversibility and irreversibility could be drawn at many other levels as well. But wherever it is drawn, for this particular application of avoiding overheating, it is hard to justify the storage requirements and/or slow-downs entailed by full reversibility, rather than just providing some small amount of cooling that would be necessary if the reversibility was only applied at a low level.

2. Information Transport. As an extension of the previous point, if our goal is just to prevent heat buildup within a microprocessor, and the heat is due to the discarding of information, then one could presumably construct a processor that did not dissipate any information internally, but rather transmitted it to some external facility for dissipation safely away from the processor. (E.g., a small radio transmitter might be very effective, if we could only get the FCC to allocate a band for energy dissipation. :-)

Such a processor might well be reversible at a low level internally, but when storing new high-level results, might transmit the old results out rather than dissipating them internally. This would achieve the goal of near-zero internal energy dissipation, but again, this would not require any of the overheads of program-level reversibility, unless there is a strong need to minimize the amount of information that is even dissipated externally.

3. Cooling. Finally, we should not overlook the possibility that the most efficient way to remove unwanted information from a processor might be to just let that information pass to uncontrolled thermal vibrational modes, but then just remove the heat rapidly by, for example, pumping a continuous stream of coolant through the device. This may seem low-tech, but in contrast with the above techniques, it doesn't require any sacrifices in terms of increased complexity or decreased speed of the circuits themselves, although they may have to be spaced a little farther apart than usual, in places, to make room for coolant channels. In any case, it is by no means clear that advanced cooling techniques will not be the preferred means of supporting high-performance processors, for an indefinitely long time into the future.

We conclude that preventing overheating is not, by itself, an application that provides a convincing justification for fully reversible computing systems.

3.1.2 Conserving Limited Energy

Aside from the heating issue, reducing power consumption of microprocessors is also of interest in applications where the energy supply is for some reason limited: e.g., battery-powered portable PC's, medical implants, microcontrollers in all manner of cordless electronic devices, CPUs in deep space probes, etc., etc.

However, some of the same arguments against using full reversibility to avoid overheating also apply to the goal of conserving energy. Partial reversibility may be sufficient to reduce the energy consumption in a particular application. And when information does get erased, this may be done by transmitting the information, this time to a special structure that is optimized to erase the information via the dissipation of the smallest feasible amount of energy.

Full reversibility does offer the enticing point that in principle a computer could perform and infinite amount of computation on a finite supply of energy, by running more and more slowly as it proceeds, but it is anticipated that this particular ability will not be too practical, as most applications will have sources of energy dissipation other than for the computation itself, and also the required slowdown will not generally be acceptable.

So again, conserving a limited energy supply does not convince us of the need for full reversibility, although again, a partial or flexible degree of reversibility may be useful. The programming paradigms discussed later in the thesis will attempt to make available a flexible range of options for how much information to discard.

So in the near term at least, low-power computing is a promising application for reversible circuit techniques, but does not yet provide much of a case for reversibility at a higher, more global level. To continue searching for such a justification, we turn to some more software-oriented sorts of applications.

3.2 Auditable/Verifiable/Trustable Computation

One interesting application for full reversibility in a computer system is to assist in meeting requirements for auditable or verifiable computation. This requirement exists in several forms for which reversibility might be useful. [Only two listed so far.]

- It should be possible to determine, with high confidence, whether a transient hardware error of some sort (e.g., a random bit-flip in memory) might have occurred during a computation, to help us determine whether the result of the computation can be considered valid.
- The system should ensure that a malicious hacker not having physical access to the hardware is unable to destroy any information stored in the system, and that the hacker's presence and complete actions can always be determined after the fact.

Let us examine how reversibility might be useful for helping to satisfy these requirements.

3.2.1 Detecting Transient Errors

Full reversibility could be used to detect transient hardware errors as follows. The initial state for the computation is set up, and we record the entire initial state, or a cryptographically secure hash function of the entire initial state. Then a computation is run, and produces some result. Then we reverse the processor direction, and run in reverse, back to the initial state. Then we compare the state with the stored state or the checksum. If there are any differences, then some error must have occurred during the computation, and the result should be considered untrustworthy.

If the processor guarantees reversibility at the hardware level, then any differences in initial state will indicate that a hardware error occurred. However, if the processor only guarantees reversibility under an assumption of the correctness of some piece of software, then differences in initial state could indicate the presence of either a hardware error or an error in that software.

Hardware errors. If the error is a transient hardware error due to some random influence from the environment (such as a cosmic ray shower, perhaps), then it is very unlikely that an error would be missed by this detection scheme, as this would require two independent error events, one during the forward computation and one during the reverse computation, that happened to cancel each other out so that the identical initial state was reached even though the final result might have been corrupted.

If such a transient hardware error is detected, then the user can simply try running the computation again, and repeat until the computation proceeds forwards and backwards with no error being detected. In this way, a very high-confidence result can always be obtained eventually, even if there is a significant probability of transient hardware errors during the computation.

This technique may also be useful for detecting the presence of some kinds of permanent hardware faults, since if the fault is permanent, then presumably the difference between initial states would presumably be the same each time. Note, however, that some permanent faults, such as a bitwise logical-complement instruction that consistently fails to flip a certain bit, will maintain reversibility, and so will not be detected by the process of comparing initial states.

Note also that we will not necessarily catch transient or permanent hardware errors if they affect the state-comparison process as well as the main computation.

Software errors. If the system does not guarantee reversibility at the hardware level, this technique will also catch those software errors that happen to lead to forward computations that fail to match the corresponding reverse computations. However, other kinds of software errors will not be caught. For example, if the system compiler correctly guarantees that all programs will be correctly reversible, then any errors in user programs compiled by that compiler will remain undetected by this technique.

Note also that if we detect an error in reversibility that is a software error, then an easy way to pinpoint its precise location is to run partial computations forwards and backwards, and find exactly how far forwards you must go before the reversibility of the system is corrupted. A binary search can be used to pinpoint the precise time of the error after $O(\log(n))$ partial computations, where n is the number of steps in the entire computation. The machine state at this time can be examined to help determine the cause of the error. This technique will be a useful tool in debugging software that is intended to ensure reversibility on hardware that does not by itself guarantee reversibility.

In summary, it seems that reversibility would indeed be useful as an error-detecting technique, although of course this will incur the usual cost of slower execution time. However, we do not know of any other technique so conceptually simple for verifying the correctness of an entire, arbitrary computation with respect to hardware failures. Thus, hardware error detection looks like a reasonable justification for full reversibility.

3.2.2 Logging/Limiting Effects of Malicious Hacking

Suppose we have a requirement that any hacker that manages to get past system security measures should (1) be unable to actually permanently destroy any data, and (2) have the complete history of his actions on the system be determinable once his interference is discovered.

A traditional approach to (1) is to make backups of data, but this does not help if the hacker destroys data before it manages to get backed up, or corrupts the backup software itself so that new data does not get backed up properly.

However, if the system guarantees full reversibility at some level that is impossible for the hacker to interfere with (e.g., at the hardware level), then by the definition of reversibility, whatever actions he takes cannot destroy any user data, or destroy any information that the hacker input to the system to do his dirty work.

Additionally, once the presence of the hacker is detected, the machine can be disconnected from the network and reversed to recover any desired earlier state of execution, and all of the hacker's actions can be observed, and the clean state prior to his break-in can be recovered.

However, note that reversibility does not protect us from the hacker's corrupting the system's outputs after the time he breaks in. He could in general still alter the running state of the system so that it produced invalid or misleading data until his interference is discovered, and its effects on the system undone. Any inputs to the system while it was in a corrupted state would have to be re-input once a clean state is restored. All those inputs could be recovered by backing up over the time after the hacker's interference.

But this suggests an alternative technique for achieving the same effects without requiring reversibility. Namely, one could simply have an incorruptible mechanism for recording the initial state of the system (when it is first turned on in a clean-slate state) and recording every bit of information that flows into the system, including any timing information, if that is important. If the system is deterministic, then that stored information is sufficient for reconstructing the complete machine state at any later time—the system is "reversible" in the sense that we could always back up to the state at any earlier time by simply going back to the initial state and proceeding forwards from there.

The obvious drawback to this technique is that if the system has been running continuously for a long time, say a year, and we only want to back up a short ways, we will have to spend another year to get up to the desired point. But then, the obvious solution is to also have an incorruptible mechanism for checkpointing the system state periodically, so that we can just go forwards from the last checkpoint.

In summary, although reversibility may be useful for tracing the activity of malicious hackers, and preventing them from damaging any data, it is not theoretically any better than just reliably recording the system's initial state and all inputs. It may or may not turn out to be easier to implement. Thus, this application is not, as yet by itself, a convincing justification for reversibility.

3.3 Database Rollback

It seems that the operation of "rolling back" the effects of an aborted transaction, which is common in some types of database systems, could possibly be implemented on top of a more general framework for undoing inter-communicating, reversible processes in a multitasking operating system for a reversible computer.

This possible application of reversibility is discussed later, in the section on Operating Systems for Reversible Computers. However, many details of this connection remain to be worked out; I can not yet say with confidence that this sort of application for reversible processing makes sense. Nevertheless, this is definitely a topic that I will explore further in the thesis work.

3.4 Speculative Execution in Multiprocessors

Similarly, it appears that reversibility might be useful to coordinate the activities of multiple CPUs which are running an underlying sequential algorithm in a parallel multiprocessing system. The individual CPUs might optimistically perform computations on data under the assumption that the data is valid (as in Knight's paper [7]), but when an inconsistency is detected, rather than restarting the processor's computation entirely, the processor might be reversibly rolled back to the point at which it read the bad data, and then proceed from there using the new, correct data.

However, I need to go back and reread Tom's paper, and I have not yet thought about this possible application carefully enough to tell whether it makes sense. It seems that it will, but I just need to work it out more carefully, and perhaps play with some simulations.

3.5 Summary of Applications

In summary, reversibility seems like it might potentially be useful for detecting some kinds of hardware and/or software errors, for keeping data safe from destruction, for saving energy and reducing heat dissipation, to some extent, and for undoing the effects of computational operations that need to be aborted, for whatever reason. Another application I have not mentioned in as much detail is to ease debugging of reversible programs.

Incidentally, John Mallery claims that there is also some interesting, deep relationship between reversibility and some issues in knowledge representation, but in my short conversations with him on this topic so far, I haven't really been able to understand what he meant; if I find the time, I should perhaps have a much longer conversation with him on this topic, and give him a chance to get me more in sync with his thought process.

However, in the end, I personally have not found any of the abovementioned near-term applications compelling enough to constitute the primary justification for the study of reversible computing, in my mind. To me, the strongest motivation is still the long-term one mentioned in the introduction, of developing computational paradigms that mesh well with the nature of the underlying physics, so that as we come up against the constraints on computation that imposed by physics more and more strongly, we will better know what to expect, and how to deal with it.

If, however, during the course of the thesis work I become convinced that some near-term application of reversibility is compelling, I will of course modify the preceding discussion appropriately.

That concludes the motivational, inspirational part of the proposal for now. Now let's get down to discussing what's going to be in the real meat of thesis.

4 Instruction Sets for Reversible Computers

[This section of the proposal, although superficially complete, does not yet convey all the details of my current thoughts on the design of instruction sets for reversible computers. I will add more details if desired.]

In the first part of the thesis chapter that will correspond to this section of the proposal, I will discuss some early instruction sets such Hall's[6] PDP-like instruction-set, Baker's Reversible Linear Lisp machine language[1, 2], and Ressler's[11, 10] Conservative Logic computer architecture.

I should include that discussion in this proposal too, I suppose, but before I do that, I really need to actually go back and look at those early instruction sets more carefully, so that I can cogently evaluate them from the point of view of building higher-level reversible programming systems on top of them. In what ways are those early architectures satisfactory in this regard? In what ways might they be unsatisfactory? At the moment, I'm still hazy on these questions, so I need to go back and study those earlier instruction sets before I try to answer.

Then, I will discuss Carlin Vieri's Pendulum architecture [13]. The Pendulum design was an important milestone in our project, but in its earlier forms it was not yet suitable as a basis for building ambitious reversible programs. I will briefly survey some minor modifications to Pendulum's architecture to which I contributed, modifications that were intended to improve Pendulum's suitability in this regard.

I will explain the need for programmer-accessible garbage data, and describe in detail Pendulum's system of forward-reverse pairs of branch instructions. I will mention some problems that are currently unresolved in the ISA, such as the inelegant current need for an extra No-Op instruction between forward and reverse jump-and-link instructions, and describe ways such problems might be resolved.

Next, I will propose (and attempt to justify) some possible extensions to the Pendulum architecture that I feel would be useful for exploring the space of possibilities for high-level reversible programming systems. These include:

Software control of the processor's direction of execution. This is useful because it allows a program to explicitly run pieces of itself in reverse as a means of disposing of garbage data. (I will say more about how this works in the section on high-level languages.) Without instruction to allow this, programs will need to keep two copies of every piece of code, one to do a forwards computation, and another to do a reverse computation. This is wasteful of memory and introduces an extra opportunity for error.

We know of several ways that the Pendulum ISA could be extended to support user control of execution direction. I propose that we pick one, or several, and add them to the tentative architecture specification and start experimenting with them.

An explicit information discarding instruction. This instruction breaks reversibility as far as programmer-visible data is concerned. Since we want to explore totally-reversible programming styles, the use of this instruction will therefore of course be optional. However, it affords us the opportunity to explore programming styles in which there is not total reversibility, and where instead we trade off reversibility against speed and memory, either at compile time or at run time, in order to meet desired constraints on memory, speed, and power.

Let us note that although the semantics of this instruction will mean that the discarded data will be unavailable to the program when it attempts to explicitly execute the instruction in reverse, the implementation will be free to implement this instruction however it wants, including via reversible storage of the garbage data in some non-programmer accessible memory (such as a continuously-running tape spool). Therefore the machine can still be said to be totally reversible for auditing purposes in case that is needed, but from the user-level programmer's point of view the machine may merely appear "mostly" reversible.

Or alternatively, the implementation may indeed discard the data, hopefully in a dissipation-minimizing fashion, in which case the system will not retain its auditability properties but may be more effective in applications where total reversibility is not an effective solution, for example, low-power applications in which there is an unlimited input stream, but limited storage facilities.

The point is that if an explicit information-discarding instruction is available, programs can be written that don't have to know about which type of system they are run on. In effect, the instruction would be a way for the program to tell the architecture, "OK, I don't care about this piece of data any more; any benefits from keeping it around are (from my point of view) not worth the cost of taking up my valuable programmer-visible storage, so do what you will with it; I won't depend on being able to reverse over this instruction to get the data back."

Again, use of the instruction will of course be optional and the thesis will explore programming styles that refrain completely from use of this instruction, as well as those that do not.

We should note that the current architecture *does* provide ways to throw data away; an illegal Pendulum program will in general do so. However, such uses are not sanctioned by the architecture spec, and may cause undesired high-energy dissipations. My proposal would be to add an *explicit* instruction which could then make a localized attempt to discard the information as cheaply as possible.

Hardware I/O Instructions. A common technique for providing I/O facilities in microprocessors is to declare that a certain range of memory addresses actually maps to a set of hardware I/O ports. I find this practice rather distasteful. I/O to external systems is conceptually a very different operation from storage of information in a system's own memory, and there ought to be, on general principles, an ISA-level abstraction for I/O operations that reflects this distinction.

One case in which the lack of a clear distinction between I/O and memory operations might be appropriate would be in a system in which all memory was declared to be shared memory that is accessible by other, independent systems. But if not all of the memory is to be shared, then either the architecture needs to make a clear statement about which parts of memory are shared and which aren't, or it needs to define a clear mechanism for specifying, at run-time perhaps, which parts of memory are shared.

As things stand, there is simply no concept of I/O in the Pendulum architecture at all. From the point of view of designing a high-level programming framework in which one could write programs that interact usefully with an external world, perhaps operating on continuous streams of input and output, this is a serious deficit.

In addition, I would suggest that the semantics of I/O instructions be defined in such a way that they could be implemented reversibly, in case the other components of the system to which our CPU interfaces are reversible as well. For example, an input instruction might XOR the input data into the target location, or require the target location to initially be clear; whereas an output instruction might specify that the output location will become clear in the course of the output operation. (In which case the output instruction could be used as a means of discarding data, discussed above—although I think there ought to still be distinct instructions for the two concepts.)

4.1 Future Instruction Sets

The final part of the section on instruction sets will propose some ideas for future ISAs that are very different from Pendulum and the other reversible instruction set architectures that have been defined.

For example, one interesting sort of instruction set would implement all instructions, including jumps and branches, in such a way that the hardware would guarantee reversibility independently of program correctness—while still allowing all garbage data to be programmer-accessible.

This is in contrast with the current Pendulum architecture, which does not guarantee reversibility unless programs are correct, and also contrasts with the original Pendulum architecture based on garbage stacks, in which, additionally, garbage was not programmer-accessible. (The original Pendulum architecture had fewer opportunities than the current one for program incorrectness to damage reversibility, but it still had some.)

The advantage of a hardware-level guarantee of reversibility would be in applications in which high or total reversibility is very important, and in which we do not wish to completely trust the compiler and run-time environment to guarantee that all programs will execute reversibly.

This might be the case if, for example, the CPU is delicate and can be damaged by internal heat dissipation, and it is very expensive, and the compiler/OS software is very complex and its correctness is hard to verify.

Alternatively, we might wish total reversibility to be preserved for purposes of auditability, to prevent accidental or intentional data destruction, or for catching and tracing the activities of malicious hackers, as discussed previously in the applications section. And yet, unless the compiler and/or OS are somehow guaranteed to be immune from any sort of accidental or intentional damage, interference, or circumvention, total reversibility will not be guaranteed. It may be much easier to simply ensure reversibility at the hardware level, rather than to try to ensure, for example, that there is no loophole in the OS or compiler that might allow a malicious hacker to cause the processor to execute code that would violate the system's total reversibility. In conventional operating systems, we often see how difficult it is to ensure the security of systems from possible malicious users who have network access, even if those users don't have physical access to the machine. Hardware reversibility could be an extra line of defense against non-physical attacks on system data integrity.

I already do know how to design an instruction set that guarantees full reversibility at the hardware level, since this is something I have thought about previously. I will present my detailed design ideas in the thesis and possibly also in a later version of this proposal. I believe the implementation to be straightforward, but I currently consider it to be beyond the scope of this thesis to implement such an architecture (except via perhaps an instruction-level software simulation), considering all the other things I am planning to do.

If I have other ideas about alternative reversible instruction sets I will present them in the thesis as well. One possibility is an instruction set for an extremely simple processing element meant to be used in a regular array of elements to form a reversible architecture similar to a dynamically programmable gate array. (For more information about DPGAs see the references at `http://www.ai.mit.edu/projects/transit/dpga_prototype_documents.html`.)

5 High-Level Languages for Reversible Computers

Once finished, this section of the proposal will discuss in detail my current thoughts on the design of high-level languages for reversible computers, and how to compile those languages down to relatively time- and space-efficient reversible programs.

The corresponding chapter of the actual thesis will develop these ideas in much more detail, and will present a complete specification for the language and the compiler. I will also present (perhaps in an appendix) complete implementations in the reversible language of the example programs described below.

First, however, I will review the few ideas for high-level reversible programming languages that have been described in the literature thus far. So far, I only know of Baker's Reversible Linear Lisp, and a reversible evaluation scheme for the lambda calculus that will apparently be presented at the upcoming Physics of Computation Conference. However, I should look back through the literature more carefully to see if any other reversible high-level languages have been described (e.g., by Ressler?).

From my point of view, one of the drawbacks of the Reversible Linear Lisp, or Ψ-Lisp, suggested by Baker in [1], is that it requires the programmer to manage all garbage data. In my language, on the other hand, I would like to explore programming paradigms that allow the programmer to write irreversible-looking code that is actually reversible because the compiler is managing the garbage data behind the scenes. Certainly an abstraction such this could be built on top of Baker's machine code. But I would like to explore this idea in a more general way, divorced from Baker's particular Linear Lisp framework, so that the language will also be suitable for compilation on other architectures, such as Pendulum.

5.1 Language Goals

These are the goals I wish to satisfy in my language design, in order of decreasing importance.

Flexibility. Since the design of high-level languages for reversible computers is still a new topic of research, we will wish to experiment with a wide range of possible language features. One way to accomplish this would be to define a different high-level language for each set of features we wish to explore. But this approach suffers from the drawback that if we wish to experiment with different combinations of features, we will end up with a plethora of languages, differing only slightly from each other. This seems inelegant, and an abuse of the term "language." If person A uses a slightly different subset of English words in his daily speech than person B does, do we really want to say that A and B are speaking different languages? What then do we name those languages?

I believe that a better way to conceptualize what we are trying to do is to define a single high-level language, one that is very broad and flexible and that accommodates all the different language features we have in mind. We will then write sample programs in that language that happen to use different combinations of language features, to experiment with those combinations. We will have a single compiler that can handle whatever constructs of the language happen to appear in a particular program. The compiler may be instructed as to which constructs to expect, to provide an independent check to ensure that a program that is supposed to refrain from using certain language features really does refrain from using them. The compiler may also be instructed as to how it should process a given construct, if we wish to explore several alternative ways of compiling the construct.

Later, once it becomes clear which combinations of language features are most appropriate and useful for writing good reversible programs, we may feel free to select particular subsets of the original language and give them independent status as languages, and name them. However, we feel that until we have gained actual experience with the various constructs possible in a reversible programming language, reifying and naming particular combinations of possible language features would be premature, and a distraction from our initial goal of exploring a wide space of language possibilities.

ISA-Independence. Although we will have particular instruction set architectures such as Pendulum in mind when designing our high-level language, we would like for most features of the language to be as insensitive as possible to the details of the ISA. The compilation process will take the original high-level program representation through a series of successive transformations, through lower-level representations that are still abstractions of the facilities that may provided by a real reversible ISA, until finally, only in the latest stages of compilation, the code is mapped to a sequence of instructions in a particular instruction set. This way, if we wish to target a different ISA, only a few particular low-level program transformations need be changed; most of the higher-level transformations will remain the same.

The thesis may demonstrate a few examples of this ISA independence, showing how some example programs would be compiled to simple alternative reversible architectures, but it is anticipated that most of the low-level program transformations I will actually write will be targeted towards the Pendulum ISA. My framework for applying program transformations will, however, be designed in such a way that it would be easy to substitute different transformations in place of the Pendulum-specific ones.

Irreversible-Looking Constructs Keeping in mind our first goal, of flexibility, one of the language features we wish to explore is the possibility of letting the programmer write high-level language constructs that look to the programmer exactly as if he were writing a program on a standard, irreversible computer, allowing the programmer to avoid having to explicitly manipulate garbage data produced during the execution of that code. Instead, the garbage data can be manipulated by the compiler behind the scenes.

What exactly to do with the garbage data can be a matter of compilation policy, or specified by other high-level language constructs. One possibility is to simply store the garbage data on a garbage stack, to allow for later reversible reclamation of that data when the program segment containing the irreversible-looking code is run backwards. Another possibility is to compress the garbage data and then throw it away using an explicit information-destroying machine instruction (if the target instruction set provides such).

Explicit Code Reversal for Garbage Reclamation. Another option that should be available to the high-level programmer experimenting with different reversible computer programming styles (in other words, to me) is to allow that programmer to explicitly run a segment of code in reverse in order to reclaim any garbage data that may have been produced by that code. There should be a high-level construct such as "reverse-call" that says, "run such-and-such code fragment in reverse." If the target ISA supports a hardware direction-reversing instruction, this construct may be implemented by simply executing the machine code corresponding to that program segment in reverse. However, if the hardware does not support this, then the compiler will need to emit two versions of the code segment, one for normal forwards execution, and another to serve as the reverse of the first segment and undo its effects.

A construct such as this will make it much easier to write programs that minimize garbage production, since it will not necessarily be possible for the compiler to figure out just what code fragments to reverse at what times to achieve the minimum dissipation. For example, given the reverse-call construct, I will be able to write a simulation of quantum mechanics (discussed later) that produces exactly zero net accumulation of garbage data over time, with only a small constant-factor slowdown. If instead I make the compiler manage the disposition of garbage data, it is not yet clear how to allow the compiler to be smart enough to do better than a logarithmic increase in space requirements over time (via Bennett's general-purpose algorithm, which will be discussed in the thesis).

Explicitly Reversible Constructs. By this I mean that the programmer should have the option to use high-level constructs that do not hide their reversibility from the programmer, but that rather provide the programmer with garbage results in a form that can be explicitly manipulated by the programmer. This feature allows the programmer to write high-level code to deal with the garbage data, such as by compressing it with some customized compression algorithm or by erasing it reversibly via another computation that the programmer knows will generate the same garbage data as the original computation.

These constructs may also be useful as an intermediate step during the compilation of the irreversible-looking constructs I mentioned earlier.

Explicit Disposal of Information. Given constructs that allow the programmer to explicitly manage garbage data, there should also be constructs that allow the programmer to explicitly return the garbage data to the compiler's jurisdiction if the program decides it no longer wishes to manage a particular piece of data.

Automatic Reversal for Garbage Reduction. I hinted at this earlier, but the language should also provide the option, either via a language construct or a compiler option, for code that generates garbage data to be executed in such a way that it appears to the programmer to just run continuously, but where actually the code generated by the compiler is broken into segments which are run both forwards and backwards in order to reclaim garbage data, in a pattern such as that specified by Bennett's algorithm [cite] or simple variations of the algorithm.

This will present an interesting challenge for the compiler, as it will require run-time support similar to that required for context-switching in a multitasking operating system. Code will be emitted that causes the entire programmer-visible program state at one point to be checkpointed, and control to be transfered to another point in the program's execution which was saved earlier. It appears to be fairly straightforward to implement this sort of facility, but there are a lot of details that remain to be worked out; however, I'm confident that the task is feasible.

General high-level language goals. Ideally, we would also like our language to support many of the features that are considered important in modern high-level languages for ordinary, non-reversible computers, such as support for structured, modular programming, abstract data types, object-oriented programming, garbage collection, type checking and type inference, parameterized types *etc.*, *etc.*

However, we recognize that, although advanced language features such as these are interesting and valuable, most of them do not have any clear, direct bearing on reversible programming *per se*, and instead represent more or less orthogonal dimensions along which any programming language could be developed. As such, these sorts of language features are not of *particular* interest to us from the point of view of the immediate goal of this thesis, which is to focus on the new issues that reversibility brings to bear on the science of computer programming. In that sense, focusing on these high-level language abilities would be somewhat of a distraction, since it is our contention that providing such features would be straightforward enough to do later after the basic framework of reversible programming is in place.

However, some basic features of high-level languages are vital enough that without them, it would be difficult to write test programs that would be large and complex enough to give us a sufficiently convincing stress-test of our basic reversible programming framework. These include such basics as:

- Nested control structures.
- Nested expressions.
- Local variables.
- Named subroutines.
- Recursion.
- Well-behaved macros.

Therefore, at least these language features, and probably some others as well, will be included in the high-level language developed for this thesis. The language will be further improved as time permits, although that work will be somewhat tangential to the main goals of the thesis.

Additionally, there are some interesting issues when a garbage collection facility, in particular, is included in a high-level reversible language. On the surface, it seems that the "garbage" of unreferenced data structures found during garbage collection might just be lumped together with the "garbage" that incrementally results from performing apparently irreversible operations, to form a single supply of garbage data that is the compiler's responsibility to deal with. But alternatively, Henry Baker has proposed a Lisp-like language with a fully-reversible language style in which it becomes simply impossible to generate an unreferenced data structure using his primitive operations.

The garbage collection issue is interesting enough that we will probably support several alternative styles of garbage-collection in our language, so that we can experiment with them, and we will discuss their pros and cons in the thesis.

5.2 Compilation Technique

With the above goals in mind, we have already made a few fundamental decisions about what our language will look like, and how it will be compiled.

The form of the source code for programs in our language will be as a single ASCII text file. We do not wish to worry about issues of putting together code from multiple sources at the moment.

Issues of lexical analysis and parsing are completely orthogonal to our goals, and therefore uninteresting, so in order to make these issues as trivial as possible, we will adopt a simple Lisp-like syntax of nested, parenthesized, s-expressions, where the first element of each s-expression is an identifying word (symbol) that determines how the rest of the expression is to be interpreted.

We will not, however, assume that we will necessarily be using any of the same identifying words that are used in existing Lisp-style languages; our choice of reserved words will be made as the language design progresses.

Compilation of programs will be performed by a program (on a normal computer) written in some powerful and convenient Lisp-style language. This will make reading in the Lisp-like source file trivial. I am intending to use Common Lisp, since I am very familiar with it, having used it a lot recently.

The first stage of compilation will simply transform the s-expression making up the source program into a slightly richer representation, as a tree of nodes whose structure corresponds to the structure of the s-expression, but where the nodes of the tree can also be annotated with additional information, at later stages of compilation.

Further compilation will proceed by applying *transformations* to the data structure, out of some set of transformations at our disposal. Each transformation will correspond to a way of implementing some language construct. The node representing the original construct will be replaced by a new node representing the result of the transformation. The different transformations will be designed to be as independent from each other as possible, to allow easy experimentation with different combinations of language features, or different ways of interpreting particular constructs.

A transformation need not take a high-level construct and map it directly all the way down to the corresponding machine code. In general, such an "all at once" transformation will not be possible. Instead, a transformation will often just map a high-level construct to a slightly-lower-level form, similarly to a macro expansion, except that the transformation may look at the context of the structure it is transforming (for example, to see how a particular local variable is bound). The high-level transformations will in general be instruction-set independent.

The lowest-level transformations will, however, produce architecture-specific results, such as register assignments, or sequences of assembly instructions in a particular ISA (instruction set architecture). The compiler can be retargeted to a new version of the ISA, or to a completely different ISA, simply by choosing a different set of low-level transformations to apply.

To make development and debugging of the compiler easier, each transformation of a structure will, in general, keep around a pointer to the old form of the structure. This will make it easy to trace back through all the transformations involved in the compilation of the program. If only the compiler was running on a reversible computer, we could achieve the same effect by running the compiler in reverse! Eventually, we would like to rewrite the compiler in our reversible language, and show that it can compile itself on a simulated reversible computer. However, although that sort of demonstration is entertaining, it is not currently considered to be a high priority for the thesis.

Incidentally, this sort of transformation-based compilation approach also facilitates compilation from different source languages (using the same intermediate-level representation), program transformation from one high-level language to another, and incremental recompilation of programs. However, though interesting, none of those directions will be explored in our compiler work, since they are tangential to our goal of investigating reversibility.

5.3 Example Programs

In this section we will look at some concrete examples of programs that I will write in the reversible programming language(s) I will design. The goal of getting these real programs working will help to keep the programming language design effort focused on producing an actual working system, and prevent the effort from getting too distracted by interesting but tangential theoretical issues in programming language design.

The following programs will be written in the high-level language, compiled using the compiler I will write, and tested, first on a simulated Pendulum architecture, and on the real Pendulum chip when it is manufactured. If there is time, I may also compile to some alternative reversible architectures, to run in simulation.

5.3.1 Simulating the Schrödinger Wave Equation

Of particular interest to us in this work will be simulations of mechanical physical systems. This is because mechanics, as we currently know it, is itself reversible.

Therefore, it ought to be possible to simulate these systems without producing any net garbage data: the only data needed to represent the current state of the computation is the computation's own representation of the current state of the physical system being simulated.

Physical systems will thus serve as an example of a significant, time-consuming computation that can be performed on a reversible computer without increasing the asymptotic time or space complexity of the computation, relative to an alternative implementation on a standard non-reversible computer.

We chose the Schrödinger equation as the physical system of interest for the following reasons:

- Quantum mechanics is inherently interesting to us from a scientific standpoint, due to the fact that it describes the atomic-scale physics that is most relevant to the exploration of the physical limits of computation. In our research we are particularly interested in following the development of quantum computers. (I may write a small section reviewing the subject elsewhere in the thesis.)
- The state of the wave function described by Schrödinger's equation is relatively straightforward to represent and iterate through time in an approximate, discrete fashion, which nevertheless behaves very nicely, illustrating behaviors such as wave-packet spreading, tunneling, and interference.
- We can implement a totally reversible version of the wave-function evolution, which behaves fairly accurately over long periods of time, without leading to ever-increasing storage requirements.

Our reversible simulation technique has already been derived, implemented in the form of a normal C program, and various aspects of its behavior observed empirically to be qualitatively correct. We briefly describe the derivation and behavior of the program here; the actual thesis will go into more detail.

To keep the simulation relatively simple, in the following discussion we will restrict ourselves to the simplest possible system, of one electron restricted to one spatial dimension, moving in a fixed (time-independent) classical potential.

However, it is simple to extend this work to make the potential be time-dependent, and we believe it will also be simple to extend the techniques described below to multiple dimensions, and multiple particles, although the space and time required by our techniques will increase exponentially with the number of degrees of freedom (like all detailed quantum simulations on non-quantum computers). It should be feasible, however, to simulate one electron in a 3-dimensional discrete space with 100x100x100 (one million) points, if that is deemed worthwhile.

Whether the thesis attempts to explore such extensions of the current work will depend on the time available.

Derivation of a discrete update rule for the Schrödinger wave equation. Here we derive a naive approximate method for simulating Schrödinger's equation in discrete, reversible fashion. Later we will derive some alternatives.

[This section probably gives too much detail. It could and should perhaps be made much more concise.]

We start with the full general form of the wave equation when expressed in a state space in which the eigenstates correspond to particle positions. It is possible to describe a system's state space in many other ways, but this way seems most straightforward and natural to those not immersed in quantum physics. It also lends itself to visualization of the wave function on a graphics display.

Here is the Schrödinger equation in its full glory:

$$-\frac{\hbar^2}{2}\sum_{j=0}^{N-1}\frac{1}{m_j}\frac{\partial^2}{\partial x_j^2}\Psi(\vec{x},t)+\mathcal{V}(\vec{x},t)\Psi(\vec{x},t)=i\hbar\frac{\partial}{\partial t}\Psi(\vec{x},t).$$

It requires some explanation for the general reader. $\hbar$ is Planck's constant over 2π, $1.055\times10^{-34}\,\mathrm{J\cdot s}$. N is the number of positional degrees of freedom, $3n$ for n particles in 3 dimensions. m_j is the mass associated with the jth degree of freedom, *e.g.* in 3 dimensions the mass of particle $\lfloor j/3\rfloor$. $\vec{x}$ is a vector of all particle position coordinates, and x_j is particular position coordinate. t is time. $\mathcal{V}$ is a potential energy function which is a function of the positions of all particles and optionally (if representing a time-dependent potential) the time. Ψ is the wave function itself, a function of the positions of all particles and of time; its value is generally a complex number with both real and imaginary parts, $\Psi=\Re\Psi+i\Im\Psi$.

The imaginary unit $i = \sqrt{-1}$ appears on the right-hand side of the equation. The magnitude of $\Psi(\vec{x}, t)$, that is, $\Re^2\Psi + \Im^2\Psi$, is, when normalized, thought of as the probability density of finding the system in state $\vec{x}$ at time t.

The important point to note, from the perspective of a dynamical simulation, is that the equation describes how the wave function evolves over time, via the partial derivative with respect to time that appears on the right side of the equation.

Now that we've seen the full form of the equation, let's proceed to strip it down to a slightly simpler form. First, we'll get rid of the summation over the $\vec{x}$ vector; since we are restricting ourselves for the moment to one particle in one dimension, there is only one coordinate x, and only one mass m.

$$-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\Psi(x,t) + \mathcal{V}(x,t)\Psi(x,t) = i\hbar\frac{\partial}{\partial t}\Psi(x,t).$$

Next, rather than writing the (x,t) arguments to Ψ and $\mathcal{V}$ repeatedly, let them be understood.

$$-\frac{\hbar^2}{2m}\frac{\partial^2\Psi}{\partial x^2} + \mathcal{V}\Psi = i\hbar\frac{\partial\Psi}{\partial t}.$$

Next, we rewrite partial derivative operators $\partial/\partial v$ as d_v/dv where d_v is a differential operator meaning "the infinitesimal change in the given quantity when variable v changes by an infinitesimal amount dv and everything else is held constant".

$$-\frac{\hbar^2}{2m}\frac{d_x^2\Psi}{dx^2} + \mathcal{V}\Psi = i\hbar\frac{d_t\Psi}{dt}.$$

This notational change will allow us to extricate the differentials in the numerator and denominator of a partial derivative operator from each other.

Now, we solve for $d_t\Psi$, the t subscript reminding us that this is the $d\Psi$ that is associated with dt in the original expression $\partial\Psi/\partial t$.

$$d_t\Psi = \frac{dt}{i\hbar}\left(-\frac{\hbar^2}{2m}\frac{d_x^2\Psi}{dx^2} + \mathcal{V}\Psi\right). \tag{1}$$

This equation gives us the infinitesimal change in $\Psi(x,t)$ at a point x as time increases by an infinitesimal amount dt and the point x is held constant. This is the most direct statement of how Ψ evolves over time.

Now we come to our first approximation. So far we have treated time and space as continuous: dt means an infinitesimally small amount of time, and dx an infinitesimally small distance in space. However, in our simulation we will *discretize* time and space into points with a minimum, non-infinitesimal distance between them, so that we only have to represent the state of the wave function at a finite number of points, and so that when we advance to the "next" time, there will be a non-infinitesimal change in the wave function state. We indicate this conceptual change by replacing d with Δ, Δx meaning the distance between our discrete points, Δt the distance between our discrete times, and $\Delta_t \Psi$ the change in $\Psi(x)$ in a time Δt.

$$\Delta_t \Psi \approx \frac{\Delta t}{i\hbar}\left(-\frac{\hbar^2}{2m}\frac{\Delta_x^2 \Psi}{\Delta x^2} + \mathcal{V}\Psi\right). \qquad (2)$$

The earlier equation with the differentials (equation 1) expresses the fact that the two sides of this equation with the deltas approach equality in the limit, as Δx and Δt approach zero. But hereafter in our derivation, we will treat the two sides as being equal even though the deltas are not zero; that is our approximation.

We believe we can prove that this approximation is correct to second order, i.e., that as long as the real rate of change of Ψ is small during the entire interval Δt, and if the minimum wavelength in the represented wave is significantly longer than Δx, then the difference between our computed $\Delta\Psi$ and the actual $\Delta\Psi$ will improve quadratically, proportionately to the square of $\Delta t/\Delta x^2$, as this ratio is decreased. The thesis will need to cite appropriate sources and justify this assertion, but we will not attempt to do so in this proposal.

Now, before we can make further progress, we need to decide how to evaluate the expression $\Delta_x^2 \Psi$ at particular points x. In general, by $\Delta_v q$ we mean the change in quantity q, a function of v, when v changes by Δv. However, how precisely are we to define $\Delta_v q(v_k)$ for a particular point v_k? One answer is that it is simply $q(v_k + \Delta v/2) - q(v_k - \Delta v/2)$.

Applying this to our particular case, we find (omitting still the ever-present t argument):

$$\begin{aligned}
\Delta_x^2 \Psi(x) &= \Delta_x \Psi(x + \frac{\Delta x}{2}) - \Delta_x \Psi(x - \frac{\Delta x}{2}) \\
&= (\Psi(x+\Delta x) - \Psi(x)) - (\Psi(x) - \Psi(x - \Delta x)) \\
&= \Psi(x+\Delta x) + \Psi(x-\Delta x) - 2\Psi(x) \\
&= 2\left(\frac{\Psi(x+\Delta x) + \Psi(x-\Delta x)}{2} - \Psi(x)\right).
\end{aligned}$$

And plugging this into the equation 2 (considered as an equality) we get:

$$\Delta_t \Psi(x) = \\ \frac{\Delta t}{i\hbar}\left(\frac{-\hbar^2}{\Delta x^2 m}\left(\frac{\Psi(x+\Delta x) + \Psi(x-\Delta x)}{2} - \Psi(x)\right) + \mathcal{V}(x)\Psi(x)\right).$$

However, rather than sprinkling things like $x + \Delta x$ throughout our equations from here on, let us take it as given that Ψ is always evaluated at points that are separated by integer multiples of Δx, and so we can treat Ψ as a vector with elements indexed by integers k. So hereafter we will replace $\Psi(x)$ by Ψ_k, $\Psi(x + \Delta x)$ by Ψ_{k+1}, *etc.*, and similarly for $\mathcal{V}(x)$ as well.

$$\Delta_t \Psi_k = \frac{\Delta t}{i\hbar}\left(-\frac{\hbar^2}{\Delta x^2 m}\left(\frac{\Psi_{k+1} + \Psi_{k-1}}{2} - \Psi_k\right) + \mathcal{V}_k \Psi_k\right).$$

Now, let's play around with this expression algebraically, and collect together the terms involving Ψ_k:

$$\begin{aligned}
\Delta_t \Psi_k &= -\frac{\Delta t \hbar^2}{i\hbar \Delta x^2 m}\left(\frac{\Psi_{k+1}+\Psi_{k-1}}{2} - \Psi_k\right) + \frac{\Delta t \mathcal{V}_k}{i\hbar}\Psi_k \\
&= -\frac{\Delta t \hbar}{i \Delta x^2 m}\left(\frac{\Psi_{k+1}+\Psi_{k-1}}{2}\right) + \frac{\Delta t \hbar}{i\Delta x^2 m}\Psi_k + \frac{\Delta t \mathcal{V}_k}{i\hbar}\Psi_k \\
&= -\frac{\Delta t \hbar}{2i\Delta x^2 m}\left(\Psi_{k+1}+\Psi_{k-1}\right) + \frac{\Delta t}{i}\left(\frac{\hbar}{\Delta x^2 m} + \frac{\mathcal{V}_k}{\hbar}\right)\Psi_k \\
&= \frac{i\Delta t \hbar}{2\Delta x^2 m}\left(\Psi_{k+1}+\Psi_{k-1}\right) - i\Delta t\left(\frac{\hbar}{\Delta x^2 m} + \frac{\mathcal{V}_k}{\hbar}\right)\Psi_k .
\end{aligned}$$

Next, to make the expression more concise, we introduce the following new quantities to use as abbreviations:

$$\epsilon = \frac{\hbar \Delta t}{m \Delta x^2},\ \omega_k = \frac{\mathcal{V}_k}{\hbar},\ \alpha_k = \epsilon + \omega_k \Delta t,$$

and rewrite our formula in terms of them:

$$\Delta_t \Psi_k = \frac{i\epsilon}{2}(\Psi_{k+1}+\Psi_{k-1}) - i\alpha_k \Psi_k .$$

But now, this is all just giving us $\Delta_t \Psi$—the change in Ψ over a time Δt. How exactly will this let us calculate Ψ at some future time given Ψ at the current time?

Well, using our earlier definition for Δ, we can expand the left hand side of the equation as follows (reintroducing t as an explicit argument):

$$\Psi_k(t+\frac{\Delta t}{2}) - \Psi_k(t - \frac{\Delta t}{2}) = \frac{i\epsilon}{2}(\Psi_{k+1}(t)+\Psi_{k-1}(t)) - i\alpha_k \Psi_k(t).$$

If we solve this for $\Psi_k(t+\Delta t/2)$, we get

$$\Psi_k(t+\frac{\Delta t}{2}) = \Psi_k(t - \frac{\Delta t}{2}) + \frac{i\epsilon}{2}(\Psi_{k+1}(t)+\Psi_{k-1}(t)) - i\alpha_k \Psi_k(t),$$

and if we now replace Δt everywhere with $2\Delta t$ (a change which does not matter since Δt was an arbitrarily chosen value already), including within the definitions of ϵ and α_k, we get

$$\Psi_k(t+\Delta t) = \Psi_k(t-\Delta t) + i\epsilon(\Psi_{k+1}(t)+\Psi_{k-1}(t)) - 2i\alpha_k \Psi_k(t).$$

Now for conciseness we replace $(t+\Delta t)$ as an argument with $s+1$ as a subscript, similarly to what we did earlier when we replaced $(x+\Delta x)$ with $k+1$, and obtain

$$\Psi_{k,s+1} = \Psi_{k,s-1} + i\left(\epsilon(\Psi_{k+1,s}+\Psi_{k-1,s}) - 2\alpha_k \Psi_{k,s}\right).$$

Well, there are a couple of important things to note about this equation. First rather than deriving the state of the wave at the next time step $s+1$ from the state at step s, the equation requires using the states at the two previous steps s and $s-1$. It's a second order difference equation, and it's of the form that Fredkin has shown to always be totally reversible, given a numeric representation that supports a reversible addition operation. (We need to cite some references on this neat trick, and explain how it works.)

The second important point is that although the equation is complex, the real part of Ψ_{s+1} depends only on the real part of Ψ_{s-1} and the imaginary part of Ψ_s. Specifically:

$$\Re\Psi_{k,s+1} = \Re\Psi_{k,s-1} + \Im\left(2\alpha_k\Psi_{k,s} - \epsilon(\Psi_{k+1,s} + \Psi_{k-1,s})\right) \tag{3}$$

and similarly, the imaginary part of Ψ_{s+1} depends only on the imaginary part of Ψ_{s-1} and the real part of Ψ_s:

$$\Im\Psi_{k,s+1} = \Im\Psi_{k,s-1} + \Re\left(2\alpha_k\Psi_{k,s} - \epsilon(\Psi_{k+1,s} + \Psi_{k-1,s})\right) \tag{4}$$

Therefore, if we consider the real components of Ψ at all the even-numbered steps $s = 2n$ (for n an integer), and the imaginary components of Ψ at all odd-numbered steps $s = 2n + 1$, we see that the evolution of those components is totally self-contained, and we can ignore the real components at odd times, and the imaginary components at even times, and thus work with only a single real number at each time step. Let us do so, and define, for integers n, the real quantities

$$\begin{aligned} \mathcal{X}_{k,n} &\equiv \Re\Psi_{k,2n} \\ \mathcal{Y}_{k,n} &\equiv \Im\Psi_{k,2n+1}. \end{aligned}$$

Then we can rewrite the equations (3 and 4) above as two equations:

$$\begin{aligned} \mathcal{X}_{k,n+1} &= \mathcal{X}_{k,n} + f_k(\vec{\mathcal{Y}}_n) \\ \mathcal{Y}_{k,n+1} &= \mathcal{Y}_{k,n} - f_k(\vec{\mathcal{X}}_{n+1}), \end{aligned} \tag{5}$$

where (for $\mathcal{Q}$ being either $\mathcal{X}$ or $\mathcal{Y}$) by $\vec{\mathcal{Q}}_n$ we mean the vector of all values $\mathcal{Q}_{k,n}$, and where (now omitting the n subscript):

$$f_k(\vec{\mathcal{Q}}) = 2\alpha_k\mathcal{Q}_k - \epsilon(\mathcal{Q}_{k+1} + \mathcal{Q}_{k-1}).$$

The update rule (5) lends itself to a particularly simple pseudo-code implementation given reversible addition/subtraction instructions, as are the C language's `+=` and `-=` operators when performed on integers:

$$\begin{aligned} \vec{\mathcal{X}} &\mathrel{+}= \lfloor f(\vec{\mathcal{Y}})\rfloor \\ \vec{\mathcal{Y}} &\mathrel{-}= \lfloor f(\vec{\mathcal{X}})\rfloor, \end{aligned}$$

where the absence of the k subscript is intended to suggest application of each operation to every element of the given vector. The n is implicit, and is no longer needed; each value of $\vec{\mathcal{X}}$ that is computed will implicitly represent the value of $\vec{\mathcal{X}}$ two time steps beyond the previous value that was computed, and each value of $\vec{\mathcal{Y}}$ that is computed will implicitly represent the value of $\vec{\mathcal{Y}}$ one time step beyond the previous value of $\vec{\mathcal{X}}$, and two time steps beyond the previously computed $\vec{\mathcal{Y}}$.

This process of updating $\vec{\mathcal{X}}$ and $\vec{\mathcal{Y}}$, can then be exactly undone by:

$$\begin{aligned} \vec{\mathcal{Y}} &\mathrel{+}= \lfloor f(\vec{\mathcal{X}}) \rfloor \\ \vec{\mathcal{X}} &\mathrel{-}= \lfloor f(\vec{\mathcal{Y}}) \rfloor. \end{aligned}$$

So the above analysis gives us a method of reversibly updating two arrays, representing the real and imaginary parts of Ψ at successive times. Note that this does not tell us both components of Ψ at either particular time, but if Ψ is changing gradually, as it will be if Δt is sufficiently small, then the two components taken together will be a fairly accurate representation of Ψ's complex value at either of the times.

We note that if there is actually only one spatial point $k = 0$, so that the "neighboring" points $k+1$ and $k-1$, are actually, in modulus 1 arithmetic, the same point as k itself, then the above algorithm actually reduces to (letting $X = \mathcal{X}_0, Y = \mathcal{Y}_0$):

$$\begin{aligned} X &\mathrel{+}= \lfloor 2\omega_0 \Delta t Y \rfloor \\ Y &\mathrel{-}= \lfloor 2\omega_0 \Delta t X \rfloor, \end{aligned}$$

which is an approximate circle-drawing algorithm, with the X's and Y's giving the coordinates of points on (or near) a circle. It is essentially the same algorithm as that described by Margolus in [9], §2.8.2, pp. 82–84, with our energy-based quantity $2\omega_0\Delta t = 2\mathcal{V}_0\Delta t/\hbar$ representing a quantity which can be thought of as being approximately the amount of change in the (X, Y) vector per $2\Delta t$ time, as a fraction of its length. This quantity takes the place of the $2\sin(\omega)$ quantity which plays the same role in Margolus's equation, determining there the angular change (in radians) of the point (x_t, y_t) across a span of 2 time steps. Note that if $\Delta t = 1$ and $\omega \approx 0$, then $2\omega\Delta t \approx 2\sin(\omega)$.

Improved algorithms. The above algorithm has the flaw that if Δt is too large, so that ϵ and all α_ks do not all remain small numbers, then the resulting evolution will be far from the sort of unitary, norm-preserving operation that we would like to have.

The dynamical system that Margolus originally described did not suffer from this problem, since his was based on an iteration $z_{t+1} = e^{i\omega} z_t$ that was exact even if ω was large. We would really prefer to have a way of stepping through the Schrödinger equation that benefited from a similar property.

I have produced other, slightly more sophisticated versions of the above algorithm, which attempt to do a better job of preserving unitarity, at least, even when Δt is large.

Actually I think no algorithm based on discrete spatial simulation will work in the case where the potential energy function that is imposed on the system leads to a particle acquiring a momentum that corresponds to a wavelength that is small compared to the separation between points. However, as long as the separation between points is much smaller than the shortest wavelength that ever appears, it should be possible to construct a simulation that is reasonably accurate even when large amounts of time are jumped over in a single step.

I could at this point go on and discuss in much more detail all my different variations of this simulations, and their pros and cons, but at this point I don't feel it's a high priority. For now, suffice it to say that I have a 100 percent reversible technique for simulating the evolution of the Schrödinger wave function in this simple system, it is accurate as a first order approximation (although I have not here taken the space to formalize and prove that assertion), and empirical demonstrations (on a normal computer) verify that the simulation behaves quite well in a variety of simple test cases involving different initial position distributions, velocities, and potential energy functions. Phenomena such as tunneling and interference have been observed. Total probability is nearly conserved. And total reversibility has been experimentally validated.

The program should be quite easily portable to a reversible processor. Updating the state need involve only integer addition and multiplication. (Pendulum does not currently support a multiplication operation, so I will need to implement one, but it should be easy to write in my high-level language.)

Therefore, I believe this program, as it stands, will constitute an interesting and feasible demonstration of a significant reversible program in my language, and will also demonstrate the ability of a totally reversible program written in my language to simulate physics without incurring an asymptotically increasing need for storage.

5.3.2 Other programs

I have no other programs, in particular, in mind at the moment. Other physical simulations would of course be quite feasible.

However, we really should also run some programs whose underlying computation is not inherently reversible, to demonstrate that a requirement for total reversibility apparently increases the space complexity of some problems. We also wish to illustrate how the $O(t^\epsilon)$ slowdown factor implied by Bennett's algorithm [5] (which can be used to keep the increase in space complexity down to a factor of only $O(\log n)$) is really not very tolerable in practice, due to a constant factor which increases exponentially as ϵ is made smaller. (This drawback was first pointed out by Levine in [8].) However, we believe Bennett's algorithm can still be useful if its use is restricted to a finite amount of compression of garbage data, followed by truly discarding the compressed data; we would like to demonstrate that as well.

For that demonstration, we should pick some standard computer science problem that has a reasonable space complexity but high time complexity (such as any NP-complete problem).

Additionally, we would like to write some programs that operate on a continuous stream of input data, to demonstrate how the ability to destroy or emit some information is necessary to prevent an otherwise-reversible computer that is operating on a continuous input stream from quickly running out of memory.

6 Reversible Operating Systems

So far I have discussed reversible computer programming from the point of view of running a single reversible program at a time. However, if we want to allow for the possibility that eventually reversibility will, for one reason or another, be in widespread use for general-purpose computing, then we should address the issue of how users might be enabled to run multiple programs concurrently on a single reversible computer, since this sort of multitasking has proven to be extremely useful in current computer systems.

Running multiple programs concurrently is traditionally part of the job of an operating system. In this thesis, we will not attempt to study how the wide range of services provided by modern operating systems might be implemented on a reversible computer, since we believe that most of these services would straightforward to implement reversibly (given our general purpose reversible-programming framework) and thus would not be very interesting.

However, what we will do in this work is to examine how a multitasking facility might be implemented on a reversible computer, since we know of some interesting problems associated with that goal.

If the multiple reversible programs are not interacting at all with each other, then it appears straightforward to run them concurrently on a reversible processor. There simply needs to be a mechanism for CPU control to be periodically transfered from within the individual programs out to an external scheduler, which decides which other program to run next, and makes the appropriate changes to machine state to transfer control to within the body of that other program, in order to continue from the point where that program left off.

One way to achieve this "escaping to the scheduler" would be to simply have the compiler periodically insert in program code instructions that transfer control to the scheduler.

In fact, even if the computer's instruction set includes software control over the direction of execution, and even if the running of individual programs involves those programs switching the CPU direction, and running pieces of their code backwards and forwards, it is still straightforward to integrate this activity with process switching, by having the scheduler keep track of which way the CPU was going when a process was last being run, as part of the saved state of that process.

However, a problem arises if we wish to allow multiple concurrently executing reversible programs to communicate with each other. Namely, what happens when a program reverses over an I/O instruction that communicates with another reversible process?

For example, suppose process A is running forwards and outputs a piece of data X through a stream-like facility, and later the datum X is received by another process B, which is also running forwards. Then process A reverses CPU direction, and attempts to undo the operating system call that sent X. How does the OS handle this situation?

Similarly, looking at the situation from B's point of view, suppose B reverses and undoes the call that received X, and then proceeds forwards again. Will the next piece of data received be X again, or will it be the next datum produced by A after it produced X?

There are several possible answers to this question.

- Disallow reversing over I/O calls. Declare such an action to be illegal, an error. However, this approach seems overly restrictive.
- Allow reversing over I/O instructions, but treat this as a no-op; don't actually undo the I/O action. This is like treating each process as if it were running on a separate reversible processor, but with an irreversible communications channel. This seems OK, but still limited.
- Finally, allow reversing over I/O instructions, but when this happens, actually reverse the flow of data in the data stream. Reversing over output is just like pulling back out the last thing that you output when running forwards; reversing over input is like stuffing the last thing you input back into the pipe.

In the thesis work we may experiment with all three options, and will perhaps think of more. The third option seems the most interesting to explore, however. It raises the further question of what happens when you attempt to reverse over an output instruction, but the program at the other end of the pipe has already pulled out and used the last datum X that you sent.

One possibility is that the program B is then forcibly reversed until it gets back to the point where it input X, so that then X gets stuffed back into the pipe, and A retrieves X (running backwards). However, this seems rude; it prevents B from doing whatever it might have been planning to do with X before reversing over the input instruction that got it.

Another possibility treats the situation more symmetrically. Suppose that reversing over an instruction to output data to a stream really is exactly like getting input from the stream—just like the stream were a totally symmetric, big, stretchy hose where you can stuff things in either end, and get things from either end, but where the things inside always remain in the same order relative to each other.

Well, then if you try to get a datum out, but there is no datum inside, then the normal thing is to *block* and wait until the data is available. *I.e.,* when A tries to reverse over its "output X" instruction, it blocks until B decides to reverse over its "input X" instruction... Just like what happens in a normal operating system when you try to input from a stream when running forwards, and there is nothing in the stream to be read.

This of course leads to still more problems. What if B pushes back into the pipe a different value than the one that A originally sent? (This can happen whenever programs have control over their own execution direction in a non-trivial way.) Program A might not be expecting to get a different value back when he reverses over his "output X" instruction; and so program A might not function as expected. (If reversibility isn't guaranteed at the hardware level, and we're not careful, program A might even fail to reverse properly, and may lose information.)

All we can say about this problem at this point is that a communications mechanism is not a communication protocol, and if we want multiple concurrent reversible programs to communicate meaningfully with each other along reversible channels such as this, we will have to describe more precisely what the intent of those communications would be. It would help if we had in mind some particular candidate application for which multitasking would be a useful abstraction.

6.1 Reversible Transaction Processing

There seems to be an interesting connection between this problem of interprocess communication in a multiprocessor, and the problem of serialization in a distributed database supporting multiple concurrent transactions.

Namely, suppose the transaction system utilizes "optimism," whereby transactions go ahead and do what they want assuming everything will be OK, but if something goes wrong and the transaction is aborted, the transaction "rolls back" and undoes everything it has done, so that a consistent database state is restored.

A problem arises when the intermediate database changes produced by that transaction have already been used (in optimistic fashion) by some other transaction. Then really, we will want that other transaction to be forcibly rolled back as well, at least to the point where it first used the bad data, so that when we let it proceed forwards again, it will be working from a consistent database state.

Well, this seems very similar to one of the approaches we proposed for how to reverse over output instructions, which was to forcibly reverse the other processes that had received the item that was output.

Thus, it seems likely that one could implement a concurrent transaction system with automatic rollback by simply building it on top of a lower-level reversible IPC mechanism that dealt with reversing over I/O operations in that way. However, many details of how this would work remain to be worked out.

Related issues crop up in Knight's paper [7], in which a mechanism is proposed for parallel execution of serial programs in multiprocessors, via a mechanism whereby a process is aborted when it is discovered to have read data that was not really valid yet at the time it was read. Rather than aborting the process, we could imagine just rolling it back to the point where the data was read, and then continuing forwards with the new value.

All these issues seem interrelated, and it seems like a single, consistent approach based on reversible communication between multiple independent reversible entities could address all of them, but the issues are still somewhat vague in my mind, and I need to go back and reread the relevant literature, until this all gels a little more firmly. In the thesis work, I will indeed make an attempt to form a synthesis of these issues, and I'm optimistic that some interesting conclusions will result, but of course the work has not been done yet, and at this point I'm still a little hazy on the issues, so I can't make any guarantees. But the thesis will report on whatever insights result from my further thoughts in this area, in any case.

To gain insight on these issues, I will implement and experiment with the various possible reversible multitasking schemes described in this section. They will be implemented using the reversible programming language described in the previous section. The thesis will describe the implementation, and present the results of the experimentation.

It is conceivable that these studies of reversible multitasking issues will lead to important new insights of immediate practical importance for multiprocessing and database systems, something along the lines of the above discussion; but at this preliminary stage I sure don't want to make any promises in that regard!

7 Reversible Physics and Computation

That about wraps up my plans for the main thrust of the thesis work, but as I stated in the abstract and introduction, one of the main motivations of all this work is to work towards the direction of making computers be more like physics in their operation so as to facilitate the eventual development of computers that take full advantage of the laws of physics. In the chapter of the thesis corresponding to this section of the proposal, I will summarize the implications of my work from that point of view, and sum up the state of the art of the physics of computation field, and state where I think things are heading, especially with regards to reversibility.

7.1 Programming Reversible Cellular Automata

This will be an important, but probably brief, section of the thesis. It will point out that although the thesis so far has focused on a serial, von Neumann-style model of computation, we believe that ultimately, to take full advantage of physics, computers will have to operate in massively parallel fashion, in which each microscopic bit of the computer is doing as much useful computational work as possible, all of the time.

Unfortunately, even when the limits of physics are not approached, the study of how to organize computational work to best take advantage of parallelism is, I feel, largely still in its infancy. However, we should say a few words about the state of affairs in that field, and about how the extra constraint of reversibility affects that picture.

Ultimately, the best parallel computational model may be some sort of reversible cellular automaton... but which one, and how do we program it to use its computational resources most effectively? This will probably be left as an open topic, but in my thesis work, I hope to at least read up on what other people have to say on this issue, and convey my impressions on what the best directions for further research will be.

In the last few weeks, I have thought a lot about how to design a universal reversible CA that would be (1) straightforwardly implementable in SCRL (split-level charge-recovery logic, the adiabatic circuit technique used in our group, see [14]), (2) straightforward to implement large circuits and programs in, and (3) self-modifiable so that it could dynamically reconfigure parts of its array to optimize them for different subcomputations. It might even be possible (depending on fabrication and packaging capabilities) to use I/O pads arrayed over the chip's whole surface area, a few per cell, so that a stack of chips could be bonded together with solder dots, allowing us to produce a scalable, arbitrarily-sized, 3-D conglomeration of cells, with the reversibility enabling the heat production in the structure to stay minimal. (The scalability comes from the fact that all the interconnections are local anyway.)

I may attempt to do a 2-D version of this CPU as my final project in 6.374 (Anantha Chandrakasan's VLSI class, see `http://www-mtl.mit.edu:80/~6.374/`) this semester, assuming that I can work out enough details of the project early enough so that it will be feasible to finish.

However, as for a 3-D version, although building a universal 3-D reversible CA in SCRL would be extremely interesting as a complement to Carlin's work on his reversible von Neumann style architecture, and would demonstrate how SCRL's power savings can allow chips to be packed together more densely, I am concerned that dealing with the fabrication and packaging issues might take too much time away from other parts of my intended research.

7.2 Quantum Computers

We in the Reversible Computing Project at MIT are keeping a close watch on developments in the field of quantum computation. Quantum computers are very interesting and relevant to us, because they are based on primitive operations (namely unitary transformations) that are reversible.

Therefore, I will probably take at least a short section of my thesis to sum up the state of the art of Quantum Computation. This should be pretty easy, since I already wrote a paper and gave a talk on the subject for my area exam, and can steal some material from that work. In that work, I even wrote a simulation of Shor's classic quantum factoring algorithm; I can include the illustrative figures produced by that program.

Additionally, if the field appears to become more promising as it develops over the next year or so, we will accordingly devote somewhat more attention to it, and I will further expand this section of my thesis to mention the latest developments.

If the quantum error correction algorithms that are currently being developed (in order to cope with the phenomenon of decoherence) turn out to be successful, we believe that the future computers that will most closely approach the limits on computational power imposed by the laws of physics will likely be some form of quantum computer. However, currently, the range of problems for which the weird parallelism of quantum computers is actually known to be beneficial is still fairly limited. That may change over the next year, since the field seems to be developing rapidly. In the thesis, I will give the reader an update on this situation as well.

If the error correction problems are solved and significant use of quantum effects does become feasible, then this suggests another extension to a physically-realistic theory of computational complexity. Such a theory would ultimately explore a model of computation based on a parallel but spatially local quantum transition rule. Preferably, the theory would also incorporate a model of spontaneous error generation. Eventually, such a model will be the only measure of computational complexity that is germane to the analysis of very large problems; other models will not realistically capture the constraints imposed by, and the opportunities afforded by, the laws of physics.

7.3 Reversible Physics versus Increasing Entropy

So far I have not had a very large number of insights on fundamental physics or computation resulting from my study of reversibility. However, here's one topic I've thought about so far which I thought was interesting, and about which I might say a few words in the actual thesis.

[Since I wrote the following section, I reread parts of Norm's thesis[9] and noticed that he has a discussion very similar to the below. Undoubtedly, my thoughts are based on Norm's discussion, since I probably read that section of his thesis last year (although I don't specifically remember it). Therefore, I need to go back through this section, and rewrite it to relate and give credit to Norm's earlier discussion.]

This deals with the interesting question of how the apparent reversibility of microphysics can be compatible with the fact that macroscopically, we see a clear distinction between the forwards and backwards time directions.

Forwards time is macroscopically associated with increasing entropy. But what does increasing entropy really mean, if the quantum evolution of the universe is truly unitary, and therefore information about the quantum state of the universe is never actually created or destroyed?

Actually, an "increase in entropy" merely means a movement of information from a macroscopic level, at which we can measure and predict and control it, to a microscopic level at which it is unobserved and uncontrolled. When a macroscopic entity dissipates, the information about its original state is still present, but it becomes all mixed up with the information about the microstate of its environment, in a very complicated way, which looks random.

In fact, this notion of increasing complexity or randomness might be formalized, which would allow forwards time to be formally defined as just that time direction which leads to increasing complexity.

Complexity or randomness can be formalized in several appropriate ways in the discrete world of computer science. It is unclear to me at present how these definitions might be meaningfully mapped to the apparently continuous world of physics. However, we can imagine that either physics turns out to be discrete after all at some level, or the definitions from computer science are somehow extended to continuous systems.

In computer science, one of the most elegant definitions of the complexity of an object is Kolmogorov complexity—the length of the shortest program that produces the given object. This metric is well-defined to the extent that it can be proven to be independent of the actual language used for expressing the program, to within a constant additive term that is infinitesimally small relative to an object's complexity, for all but an infinitesimally small fraction of objects.

If this concept could be mapped onto physics, then presumably the shortest program that could print out the complete state of the universe at some time would be a program that just applied the laws of physics to a simple initial state, with the time given as an input. The length of the input is included in the Kolmogorov complexity, and so the total Kolmogorov complexity would generally increase (logarithmically) with increasing time. This would give us a metric of time based on microstates.

Note that if the physics is reversible and happens to have a finite number of reachable states, eventually the state of the universe will converge back onto the initial state—and while the initial state is being reached, the shortest description of the time input will be a negative number counting up to zero, and so the complexity will (by the Kolmogorov definition) be decreasing, and so at that point time might be defined to be running backwards. I.e., if forwards time is defined as increasing complexity, then time goes forwards as you move away from the presumed simple initial state, no matter which way you go along the bidirectional chain of states. Since microphysics is apparently time-symmetric (with corresponding reversals of charge and chirality, if I am remembering correctly), we would expect that the evolution in either direction from the initial state looks pretty much the same. So if the universe dissipates into a cloud of gas, but then eventually recondenses in a way that somehow recreates the exact initial state of our big bang, say, then we may expect that during the recondensation process there will be beings like ourselves, but moving backwards and thinking backwards from our perspective; and clouds of steam condensing into teapots, etc.

We may point out one flaw in the above Kolmogorov definition of universe-state complexity, which is that a program may be used to represent the time input as well. Therefore, "physics after 10^{30} steps" will be declared a "less complex" and therefore "earlier time" state than "physics after 3945871938190 (*i.e.*, some random number) steps" just because the time is shorter to express, even though its value is larger.

We can correct this flaw by instead using Charles Bennett's alternative measure of complexity, which is the *running time* of the shortest program that generates a given object. With this measure, universe "complexity" increases monotonically as you move away from the simplest (initial) state in either direction—except if the universe happens to go through an intermediate state that happens to have a shorter exact description than the initial state, the physics, and the number of steps all taken together. This may be deemed unlikely.

However, we may note an interesting exception in the case where the chain of states is finite. In this case, the universe may spend a very long time in states that look essentially random, corresponding to the "heat death" of the universe. The interesting thing about Kolmogorov complexity is that the shortest description of a "random" state is a program that just prints it out. And printing out a "random" state is going to be much faster than getting to the state by iterating through possibly exponentially-many predecessor states in a physical simulation. And also, printing any random state of a given length takes the same amount of time.

So during the extremely long "heat death" period of the universe, there would, by this metric, be no change in time at all, since all of the random-looking states would have simplest descriptions which would be to just print the random-looking state, which would take about the same amount of run-time for any of them. The value of the time metric would be relatively small, expressing naturally that any of those random states looks pretty primordial.

In summary, entropy as "loss of information" refers really not to information loss in a deep physical sense but rather to a complex mixing-up of the information, to the point where the information is no longer present at the relevant level of description of the system that we are interested in. And the fundamental reason why we generally consider the state of particular thermal modes to be irrelevant is that it will take an astronomically long time before the minute details of the thermal state of the system might be expected to converge to produce an interesting macroscopic effect. (Such as atoms spontaneously converging to form a regular macroscopic structure.)

One interesting interpretation of the way we interpret and perceive the randomization of the state of the universe as an increase in time, is that this corresponds to the increase in running time of the shortest program that describes the current state.

Or rather, perhaps we should say that in the formal notion of a program that starts with an initial state and maps it to a complex sequence of later states, we perceive this program as "running forwards" simply by analogy with our own experience with a world that randomizes (complexifies) itself, a process we call "running forwards in time."

7.3.1 So, what will the thesis say about this topic?

Unfortunately, I don't know if, in the actual thesis, I will have time to do much more than speculate along the lines above. However, if people are interested, I can survey Kolmogorov and Bennett complexity and couch the above discussion in more rigorous terminology. Any suggestions for ways to pursue this line of thought further would be appreciated.

7.4 Any more Physics and Computation directions planned for the thesis?

Nothing concrete planned at the moment. I very much enjoy studying and learning about the physics of computation, but I don't think I have enough hard-core physics background yet to really contribute any fundamental new insights to the field. If I do think of something, though, I will certainly let everyone know!

8 Conclusion

Of course, I cannot yet say what the actual thesis will conclude, but I can say that I believe the research program outlined above is feasible, original, and interesting, and will produce substantial enduring results in the form of languages, compilers, and multiprocessing techniques for reversible computers, and may turn out to be an extremely valuable piece of work, if any of the possible applications of reversibility eventually do develop to fruition. Additionally, I hope to establish in my work that some near-term applications for reversible computers may be reasonably expected to be useful, and programming these computers, at least, is straightforward using the techniques I will explore.

In any case, the work will still contribute to the ongoing quest of science, of extending the realm of human knowledge. Moreover, it seems that our civilization's demand for computational power is completely unlimited, so that eventually, if technical and economic obstacles are overcome, there will eventually be a drive to build computer systems that take full advantage of every cubic Angstrom of available space and every electron-volt of available mass-energy within the computer in order to achieve maximum speed. At this point, we believe that the need for the computer to manipulate data in a primarily reversible fashion will be essentially inevitable. It is hoped that when the computer scientists of future centuries are finally forced to face this fact, they will find this thesis a valuable reference on, or at least a significant historical development in, the study of programming reversible computers.

References

[1] H. G. Baker. NREVERSAL of fortune — the thermodynamics of garbage collection. In Y. Bekkers, editor, *International Workshop on Memory Management*, pages 507–524. Springer-Verlag, 1992.

[2] Henry G. Baker. Lively linear lisp—'look ma, no garbage!'. *ACM Sigplan Notices*, 27(8):89–98, August 1992.

[3] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 6:525–532, 1973.

[4] C. H. Bennett. Notes on the history of reversible computation. *IBM Journal of Research and Development*, 32(1):281–288, 1988.

[5] C. H. Bennett. Time/space trade-offs for reversible computation. *Society for Industrial and Applied Mathematics Journal on Computing*, 18(4):766–776, 1989.

[6] J. Storrs Hall. A reversible instruction set architecture and algorithms. In *Physics and Computation*, pages 128–134, November 1994.

[7] Tom Knight. An architecture for mostly functional languages. In Patrick Henry Winston and Sarah Alexandra Shellard, editors, *Artificial Intelligence at MIT: Expanding Frontiers*, volume 1, chapter 19, pages 500–519. The MIT Press, Cambridge, Massachusetts, 1990.

[8] Robert Y. Levine and Alan T. Sherman. A note on Bennett's time-space tradeoff for reversible computation. *Society for Industrial and Applied Mathematics Journal on Computing*, 19(4):673–677, 1990.

[9] N. H. Margolus. *Physics and Computation*. PhD thesis, MIT Artificial Intelligence Laboratory, 1988.

[10] A. L. Ressler. Practical circuits using conservative reversible logic. Bachelor's thesis, MIT, 1979.

[11] A. L. Ressler. The design of a conservative logic computer and a graphical editor simulator. Master's thesis, MIT Artificial Intelligence Laboratory, 1981.

[12] Tommaso Toffoli. Computation and construction universality of reversible cellular automata. *Journal of Computer and System Sciences*, 15:213–231, 1977.

[13] Carlin J. Vieri. Pendulum: A reversible computer architecture. Master's thesis, MIT Artificial Intelligence Laboratory, 1995.

[14] S. G. Younis. *Asymptotically Zero Energy Computing Using Split-Level Charge Recovery Logic*. PhD thesis, MIT Artificial Intelligence Laboratory, 1994.