Patchable Instruction ROM Architecture

Timothy Sherwood   Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114

{sherwood,calder}@cs.ucsd.edu

ABSTRACT

Increased systems level integration has meant the movement of many traditionally off chip components onto a single chip including a processor, instruction storage, data path, and local memory. The design of these systems is driven by two conflicting goals, the need for reduced area and the need for rapid development times. The two current design options for instruction storage, ROM and Flash, are each highly optimized to one of these two goals but provide little compromise between them. ROM is used for highly area optimized instruction memory, although this comes at a price of lengthy integration time due to it's need to be correct before the chip is sent for fabrication. Flash is an alternative instruction memory that can significantly reduce the time to market by allowing embedded software to be upgraded after fabrication, meaning that software test and fabrication can be overlapped. Unfortunately Flash takes over a factor of 2 times the area of the equivalent ROM based storage.

In this paper we present the *Patchable Instruction ROM* as an architecture for instruction storage that can provide the best of both worlds – reduced area and faster time to market. With area efficiency similar to a standard ROM and support for limited post fabrication software patching, Patchable Instruction ROM provides a new set of design points to consider when building embedded systems. For the programs we examine, we show that our hardware/software technique can achieve an area only 10% larger than ROM with only an 11% inflation in design time over a Flash based approach.

1. INTRODUCTION

The rapidly growing embedded electronics industry demands high performance, low cost systems with increased pressure on design time. Now more than ever there is increased pressure on designers to get their systems out the door as quickly and as efficiently as possible. Many designers are turning to embedded microprocessors as a way of meeting these increasing design time pressures.

In order to feed these microprocessors, instructions must be provided in a form of non-volatile memory. To save on packaging and pincount, a trend is for this non-volatile memory to be integrated into on-chip ROM. The HP DeskJet 820C digital controller ASIC [9] is a perfect example of this design trend. On the ASIC, the data path is implemented in standard cell, a microprocessor is included for control, and and ROM is included to direct the microprocessor. The area taken up by the ROM is 14% of the total die area, almost as much as the microprocessor, and the sizes of embedded memory will continue to grow as embedded processors continue to become faster. If the memory on the 820C controller were instead implemented in Flash it would have taken up 25% of the die area.

Building such a chip with ROM requires that all ROM testing be completed before the final fabrication of the chip is released. As each generation of embedded system develops, more and more complexity is expected to be dealt with by the software of the system. This means that with each successive generation of chip more software will be embedded into on-chip ROMs.

Since the ROM code needs to be hardwired into the system, the testing of the system software needs to be serialized with fabrication of chip. This means that the final fabrication has to wait for testing of the ROM code to be complete, and this can significantly increase the time-to-market. The top of Figure 1 shows this graphically with the dotted line showing the time the chip can be released. This is the total time the code is tested in order to produce the ROM code, plus the fabrication time that starts after testing has been completed. Custom fabrication has a very long lead time, on the order of 90 days [4]. This means that it would be very beneficial to provide an overlap of the final stages of test with the actual fabrication of the chip.

An alternative to using ROM is to use a *writable* non-volatile memory. Flash is an alternative instruction memory that is more than twice the size (and cost) of mask-programmable ROM [9]. Flash allows a full code rewrite after the chip has been fabricated as long as the code size does not grow larger than the Flash that has been allocated. This allows the code testing and the chip fabrication to be overlapped as shown in Figure 1. In this example, testing time was chosen to be larger than fabrication time, therefore the chip can be released after testing has completed and the final code is written to the Flash. This is designated by the dashed line after the testing time. This results in a faster time-to-market, but comes at the price of increased area and cost.

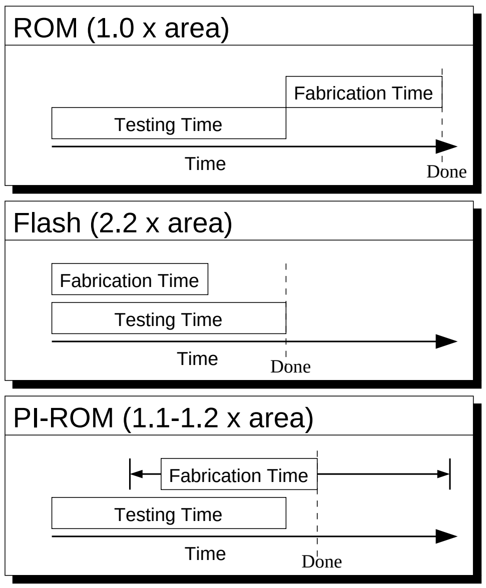

Figure 1: The three options that are available to the embedded system designer for instruction memory. This first option is to use a ROM. ROMs have the highest density, but require the code to be fixed before fabrication begins, and thus require that code testing and fabrication be serialized. Flash is the other extreme, allowing full post fabrication configuration at the cost of bloated area compared to ROM. Using a Patchable Instruction ROM can overlap most of the testing time with fabrication with only a small increase in area.

We would like to design a system that has the cost advantages of integrated ROM, while still maintaining the flexibility of FLASH. The approach we propose in this paper aims to overlap the long tail of test with the actual fabrication of the chip. We will take advantage of the fact that relatively minor changes to the code after the initial phase of testing are needed, and these could be captured by a *small* amount of writable memory. In order to do this we need a way to provide limited post-fabrication customization without seriously impacting the cost of the chip.

To this end we present the *Patchable Instruction ROM* (PI-ROM) architecture, and evaluate it for patching a fabricated ROM with bug fixes and software updates. The patch is provided in hardware by a small writable *Patch Memory*. A jump table is used to direct the fetch PC from the ROM to the Patch Memory to execute the patch code, and execution returns back to the ROM when the patch has finished executing. This architecture allows the system designers to aim for a more aggressive fabrication schedule knowing that small changes to the code will still be possible after returning from fabrication. We show how new versions of the code can be compiled with minimal size patches and how the architecture uses these patches to provide the correct instruction stream. We further evaluate the design tradeoffs involved, and how one would pick a point in the design space that will provide small chip area along with a high probability of full update coverage.

2. ARCHITECTURE

In order to have efficient storage of instruction memory we need to make use of the high density provided by on-chip ROM, without suffering increased time-to-market. Fabrication needs to overlap testing in order to meet these time constraints, but this requires the ability to modify the code after fabrication. Our solution trades off a small portion of the density achieved by using standard ROMs to increase post-fabrication flexibility by allowing patches to the code to be added after fabrication. We now describe an architecture and compiler approach that can be used for this purpose.

2.1 Bene£ts of Patchable Instruction ROM

The manner in which software defects are discovered and fixed is a well studied art in production software development and by the software engineering community. There is diminishing returns over time as one attempts to fix more defects in the code. The majority of the defect fixes come fairly early in the testing phase, but there is a long drawn out tail where extensive testing is needed to get the system to the desired level of correctness. While this tail of testing is necessary to keep product standards at reasonable levels, it statistically yields very few bugs. We use this fact as the underlying motivation for our architecture.

The *Patchable Instruction ROM* (PI-ROM) architecture uses a ROM for the majority of the code storage, but still allows post fabrication customization using a small amount of non-volatile writable memory and lookup structures. The PI-ROM is fabricated after an initial testing phase, where the majority of the software defects have been fixed. While the developers are waiting for the chip to come back from fabrication they continue to further test and fix defects in the embedded code. When fabrication is done, patch code is placed into a very small non-volatile writable *Patch Memory* on chip to achieve the desired code quality. At a high level the PI-ROM architecture operates by executing code in the original fabricated ROM and switching over to the Patch Memory when a defective section of software is encountered. Execution will then switch back to the ROM after executing the patch code. Using this scheme we are able to overlap most of the fabrication time with the tail end of code testing, while only increasing the area by a small amount.

The bottom of Figure 1 shows how the PI-ROM architecture allows a variable amount of overlap in testing and fabrication. Fabrication starts after a sufficient amount of testing time has occurred to eliminate a majority of the defects. The shorter the fabrication time is in relationship to the testing time, the more time there is for testing before fabrication needs to start, and the smaller the Patch Memory will have to be. In contrast, if the amount of time for fabrication is close to or larger than the testing time needed, then the tail part of the fabrication will not be overlapped with testing.

2.2 Hardware

The PI-ROM architecture uses three distinct structures as can be seen in figure 2. The first and largest structure is the main Instruction ROM. The ROM stores a version of the software code that has been well tested, but is not quite ready for final release. When the code has reached an acceptable level of quality (to be examined in section 3.4), the code is sent along with the other hardware descriptions of the chip to start fabrication, and this version of the code

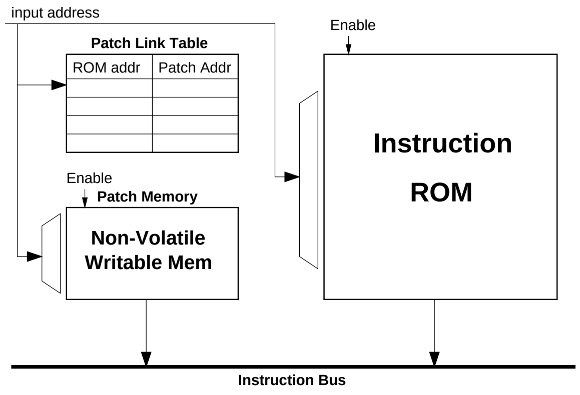

Figure 2: The Patchable Instruction ROM architecture is comprised of three structures, a ROM that stores a preliminary version of the code, a very small Flash or other non-volatile writable memory that stores the patches to the preliminary version, and a Patch Link Table that provides the mapping between the two at run time.

is stored in the ROM.

While the developers are waiting for the chip to come back from fabrication they continue to further test and fix defects in the embedded code. During this time the designers come up with a "final" version of the code. This version will contain software defect fixes resulting in small changes to the version of the code stored in the ROM. These software defect fixes will be stored as small code patches in the Patch Memory, an on-chip writable non-volatile memory (e.g., Flash or EPROM) as shown in figure 2. After the chip comes back from fabrication, it can be programmed with the proper patch code to provide correct execution.

A *Patch Link Table* (PLT) is a small fully associative table that stores all of the ROM program counter locations that are followed by the start of a patch. This forms the bridge between the code in ROM and the patch code. The associative lookup into the PLT checks to see if the current instruction is in the table. If it is, the PLT returns the location of the patch in the patch code address space and this becomes the next fetch address. The number of bits for a single entry in the PLT scales as the sum of the log of ROM address space and the log of the patch address space.

When the Instruction ROM is accessed, the Patch Link Table is checked in parallel. If there is a hit found in the PLT, then the *next* instruction fetch will come from the specified location in the Patch Memory. In this way, instructions that have addresses found in the Patch Link Table act as an *implicit jump* to the patch code. The last instruction of the patch will jump back into the address space of the ROM. This will make the next instruction fetch to be taken from the ROM. There is no latency in transitioning from the ROM to the Patch Memory, since a hit in the PLT will redirect the fetch in the next cycle to the Patch Memory.

In this paper we assume an architecture without an instruction cache, which is representative of most of the embedded microcontroller domain. Even so, the use of an instruction cache integrates seamlessly into our PI-ROM architecture. Each fetch address for the I-cache lookup would be looked up simultaneously in the I-cache and the PLT. If a hit is found in the PLT, then the next fetch address is changed to be the translated (patch) address from the PLT. This is the identical behavior as described above for transitioning from the ROM to the Patch Memory. The only reason this could change the overall behavior of the system is if the Patch Memory mapped to an area of the cache that was conflicting with other instructions in the surrounding code. However, because we can control the area to which the Patch Memory maps we can perform conflict analysis to insure that this does not happen.

2.3 Patch Compiler Support

A new form of compiler support is needed to generate the patch code and the Patch Link Table mapping. This is to allow the patch code to correctly interact with the original pre-fabrication code in the ROM.

After testing has completed, the *final code* contains minor differences from the *pre-fabrication code* stored in the ROM. The goal of the compiler is to detect these minor differences, and to form regions of code that represent patches. Determining differences between the two code baselines can be done at many different levels. These range from comparing source code, intermediate representations, or binary level differencing. Determining differences between two code baselines is a well studied problem in industry, and there are many tools that provide this ability. In section 3.4, we gathered our results by generating patches from the binary level.

Once the compiler has determined the sections of code that are different, it needs to form the patch regions. A *Patch Region* is a grouping of sequential instructions that will be executed in the Patch Memory instead of ROM. Each patch region uses one entry in the Patch Link Table. The Patch Link Table is architected to be small in order to optimize area on the chip. In this paper we assume a fixed number of PLT entries, which are provided to the compiler to work with.[1] Since the compiler cannot generate any more patches than PLT entries, it may need to combine several code differences with spatial locality into one patch region. The compiler will generate regions until all of the patches can be stored in the PLT, and the code regions fit into the Patch Memory.

If there is no way to fit all of the necessary code updates into the Patch Link Table and Patch Memory, a respin of the chip may be necessary. For a pure Flash solution, a similar problem can occur. If the amount of Flash memory chosen to be fabricated on the chip for instruction memory ends up being too small to hold the final code image, a new spin of the chip may be needed. For a pure ROM solution, a new spin of the chip would be required if a severe defect is found in the code after fabrication.

After the patch regions are formed, they are compiled to interact appropriately with the ROM code. This consists of (1) maintaining the correct register mapping, (2) maintaining the correct data addresses, (3) identifying the addresses where the ROM jumps to Patch Memory, which will be stored in the Patch Link Table, and (4) identifying what address each patch will jump back to in the ROM when it is finished executing the patch.

[1]The Patch Link Table could potentially be made to be dynamically loaded depending upon the region of ROM code being executed in order to provide more entries, and this is left for future work.

Original Code

0x180	addq	a2, 0x1, a2
0x184	addq	a0, 0x1, a0
0x188	addl	t4, 0x1, t4
0x18c	br	zero, 0x1b4
0x190	bne	t7, 0x1a0
0x194	bis	zero, zero, v0
0x198	br	zero, 0x1c0
0x19c	bis	zero, zero, zero
0x1a0	subl	t9, t4, t0
0x1a4	bis	t7, t7, a2
0x1a8	zapnot	t0, 0xf, t0
0x1ac	bis	zero, zero, t4
0x1b0	addq	a0, t0, a0
0x1b4	cmpult	a2, t5, t0
0x1b8	bne	t0, 0xe0
0x1bc	lda	v0, 1(zero)
0x1c0	ret	zero, (ra), 1

Recompiled with update

0x180	addq	a2, 0x1, a2
0x184	addq	a0, 0x1, a0
0x188	br	zero, 0x1a0
0x18c	bis	zero, zero, zero
0x190	beq	t3, 0x1b4
0x194	addq	t7, 0x1, a0
0x198	bis	t3, t3, a2
0x19c	bis	a0, a0, t7
0x1a0	cmpult	a2, t5, t0
0x1a4	bne	t0, 0xe0
0x1a8	cmpult	a0, t4, t0
0x1ac	beq	t0, 0x1e0
0x1b0	bne	t3, 0x1c0
0x1b4	bis	zero, zero, v0
0x1b8	br	zero, 0x1e4
0x1bc	bis	zero, zero, zero
0x1c0	bis	t3, t3, a2
0x1c4	subq	t5, a2, t0
0x1c8	bis	zero, zero, t3
0x1cc	cpys	$f31,$f31,$f31
0x1d0	subq	t4, t0, a0
0x1d4	br	zero, 0xc8
0x1d8	bis	zero, zero, zero
0x1dc	ldq_u	zero, 0(sp)
0x1e0	lda	v0, 1(zero)
0x1e4	ret	zero, (ra), 1

Patch Region

0xP00	br	zero, 0xP18
0xP04	bis	zero, zero, zero
0xP08	beq	t3, 0xP2c
0xP0c	addq	t7, 0x1, a0
0xP10	bis	t3, t3, a2
0xP14	bis	a0, a0, t7
0xP18	cmpult	a2, t5, t0
0xP1c	bne	t0, 0xe0
0xP20	cmpult	a0, t4, t0
0xP24	beq	t0, 0x1bc
0xP28	bne	t3, 0xP38
0xP2c	bis	zero, zero, v0
0xP30	br	zero, 0x1c0
0xP34	bis	zero, zero, zero
0xP38	bis	t3, t3, a2
0xP3c	subq	t5, a2, t0
0xP40	bis	zero, zero, t3
0xP44	cpys	$f31,$f31,$f31
0xP48	subq	t4, t0, a0
0xP4c	br	zero, 0xc8
0xP50	bis	zero, zero, zero
0xP54	ldq_u	zero, 0(sp)
0xP58	jump	0x1bc

Patch Link Table

0x184	0xP00

Figure 3: The function `string_match` from gs, shown with fault, in a corrected version, and the patch that would be generated from the correction. The instructions highlighted required modification to fix the defective software. The addresses for the patch region are shown in the form `0xPnn` to show that they are distinct from the addresses contained within the ROM. Note how the branch targets must change in the patched version, and how the jump in inserted at the end to link back to the original code.

2.4 A Patch Code Example from Ghostscript

In this section we will examine one of the code patches for the GNU program ghostscript. We will take a baseline distribution of the program, and examine generating patch code by applying a distributed patch to the original program. We used version 2.6.1-0 as the baseline version of the distribution, and version 2.6.1-1 as a patch release that fixes 5 defects present in the original version. We examine the correction of a defect in the function `string_match`, which is used to match strings including wild cards. In the original version, the function did not perform correctly on strings of different length. The fix affected 7 different lines of code.

The tail end of the function required changes in both the control flow and the data flow of the function. The original assembly code for the latter part of the function is shown in Figure 3 in the table labeled `Original code`. The instructions highlighted required modification to fix this defect. The code labeled `Recompiled with update` represents the same part of the function recompiled into a binary with the patch fix applied. The highlighted area shows the changed area of the code, between the two binaries.

The right part of Figure 3 shows the binary for the patch code to be stored in the Patch Memory to fix this problem. The patch addresses in the figure are shown of the form `0xPnn` to show that they are distinct from the addresses contained within the ROM. The first thing to notice is that the Patch Link Table must be instructed to jump to the patch region following the fetch of instruction at address `0x184`. If there are any other changes of control flow that target the region of original code containing the defect these too must included into a patch region and patched.

Branch targets must be linked to the correct locations in either the patch memory or back to the ROM. If the branch was to an address within the patch region, then it must now target the patch memory. This can be seen in the instructions at addresses `0xP00`, `0xP08`, and `0xP28` of the patch memory. In addition, addresses that link back to the code in ROM need to be remapped. The `bne` at `0xP24` is a prefect example of this, its resulting target is `0x1bc` back to the original code in the ROM.

3. EVALUATION

In this section we evaluate the time benefits of PI-ROM compared to a traditional ROM based approach, and the area benefits of the PI-ROM over using Flash memory to store all of the code.

We evaluate the effectiveness of the PI-ROM architecture by examining three real programs `ghostscript`, `sleepy cat`, and `zlib`, each with at least two versions of the code provided by their respective authors, an initial release and one with bug fixes. The program `ghostscript` is a postscript interpreter, `sleepy` is a database built for use in embedded systems, and `zlib` is a compression library. From the initial version and updates we gathered the size of the production release, the number of the necessary patch regions, and the average size of the patch regions. Update releases that included extra features or included patches to port the system were avoided if possible.

3.1 PI-ROM and Flash Area Tradeoffs

We first provide area model results used to evaluate the tradeoffs between ROM, PI-ROM and Flash. To estimate

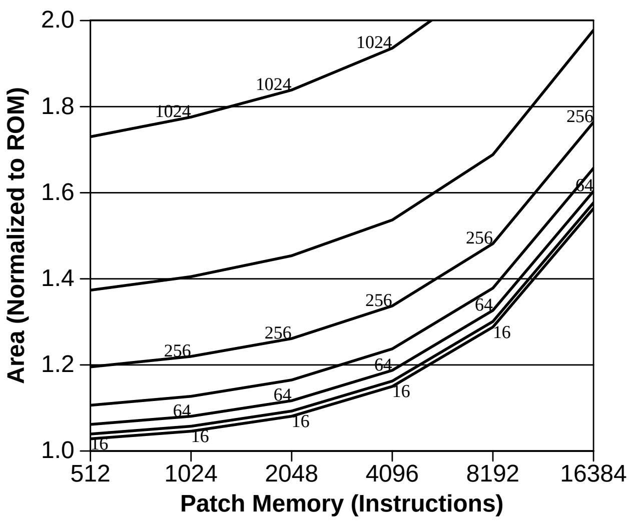

Figure 4: Area of the instruction storage architecture with a ROM size that can hold 64K instructions versus the amount of patch code that can be supported. Each line represents a different number of patches that are supported. For example, a PI-ROM that can store 4096 instructions of patch code in 64 distinct patches will take up approximately 20% more area than a 64K ROM.

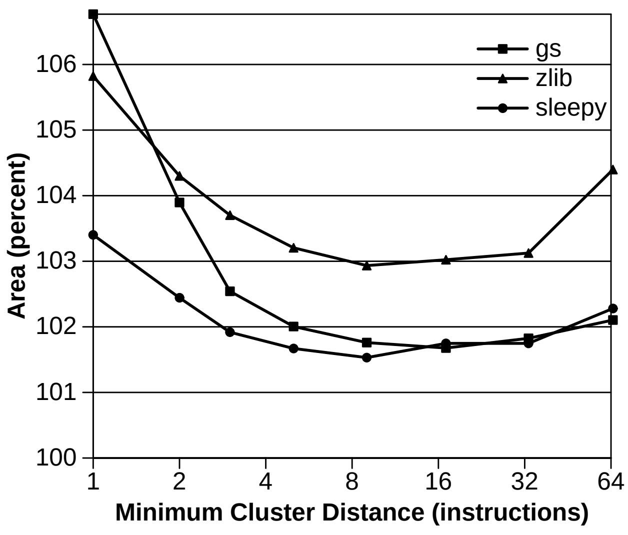

Figure 5: Total Area versus Patch Cluster Width. As we merge clusters of different instructions together into patch regions from different distances we can see the effect it has on the area. If we do no clustering the area of the Patch Link Table increases the total area. If instead we are too aggressive in clustering for patches we include too many redundant instructions and the size of the Patch Memory grows too large. The optimum cluster distance for the three programs we examined is between an 8 and 16 instructions.

the area of the three different structures, we first estimate the number of bits necessary for the structure and then multiply that by the per bit density of that memory structure. All three memory types are common VLSI structures with a high degree of regularity. We calculate area from recently publish work [13, 14, 11].

The PI-ROM architecture makes use of both ROM and Flash technology. In this architecture, ROM is used for the majority of the code, while the Flash can be used for the Patch Memory. In addition, we use a fully-associative lookup structure for the Patch Link Table. We assume that the lookup table is built from a volatile memory technology, similar to a cache. For this reason an additional small Flash is used to store the contents of the Patch Link Table for when the system is powered down. At power up time the Patch Link Table is initialized with the contents of the appropriate section of Flash.

In examining recent circuit implementation papers, we see that each ROM bit has an area of approximately $4f^2$ [13] where f is the minimum feature size. Flash densities are between $8f^2$ and $10f^2$ [14]. This is a large enough difference that simply building the entire memory out of Flash may not be cost effective. In contrast, adding a small amount of extra area for the Patch Memory and the PLT store will not have too much effect on costs. Fully associative table densities on the other hand, lag far behind the other two because of the logic required to do the comparison, and tend to be around $200f^2$. It is because of this density difference between the associative lookup table and Flash, that it is usually more area efficient to combine two or more patch regions into a single patch region, as described in section 2.3, even if it involves including instructions that do not need to be patched.

Using these densities we calculate how many bits are in each structure to compare their total areas. The ROM and Patch Memory are memory banks, and the number of bits needed is given as the number of instructions contained in the memory multiplied by the width of the instruction used. For the rest of the paper we assume an instruction width of 16 bits. The Patch Link Table, as was mentioned in section 2.2, needs to store two numbers, the ROM address to trigger the jump to Patch Memory, and the target address in the patch memory address space. The total number of bits for the patch link table is given as:

$$PLTbits = (\log_2(Size_{ROM}) + \log_2(Size_{FLASH}))N_{patches}$$

Figure 4 shows the area of the PI-ROM instruction storage architecture as you vary the number of PLT entries available for patch regions. The areas shown are relative to a single program ROM of 64K. Note that for this configuration if we are willing to grow the instruction storage by 20% over a ROM we can support 4 Kbytes worth of patch memory with up to 64 distinct patches. If the number of instructions per patch is typically lower than this, we can increase the number of patches and decrease the total patch code size or the opposite. When the PI-ROM architecture reaches 200%, its area is at the same level as an architecture using only Flash for its instruction memory.

3.2 Examining Patch Characteristics

For the results gathered in this paper, we form the regions by performing binary matching between the production release and the patch release of each application. Section 2.3

described the steps a compiler would take to form the regions. In this section, we describe the steps we take to create a reasonable estimate for the code changes and where they are, without implementing a full compiler solution.

To determine the number of changed lines of code from a release to a patch, we start by taking each program and compiling it to an Alpha binary. We then apply a set of patches provided by the application writers, and recompile the application to a patch binary. We then compare these two binaries to find the sections of code that have changed. To compare the two binaries, we ignore register names, data addresses, and branch addresses, since these may have arbitrarily changed during the two compilations. Examining the version with everything else left provides a very good estimate as to the exact number of lines of code that have changed, and where those changes are. [2] In addition, if there are conditional branches in the original code that jump into a region of code that is patched, these branches must also be patched in order to jump to the newly generated patch code.

The next step in generating the patch memory binary is to find patch regions that provide a good tradeoff between patch size and the number of regions. If we allow no redundant instructions in the ROM and Patch Memory, then each contiguous set of different instructions is a separate patch. This provides the smallest ROM but requires a very large Patch Link Table to capture all of these individual small patches. If instead the compiler chooses to merge patch regions together, even though the changes are not contiguous, we reduce the area of the Patch Link Table at the cost of a larger Patch Memory.

To examine these changes we begin with a longest common subsequence analysis of the two different executables and find changes between the two. This provides us with a set of patches that are provably as small as possible. We then begin merging patches together. We merge two patches together into a single patch region if their distance (number of instructions between the end of the first patch and the start of the second patch) is less than a threshold. Figure 5 shows the effect of varying the value of the threshold from 1 to 64 instructions. The minimum point on the graph is at a cluster distance between 8 and 16 instructions. This is to be expected, since each entry in the Patch Link Table we examined is about the same size as about 14 instructions stored in Patch Memory. At the minimum point in the graph we have a balance between increasing Patch Memory redundancy and reduced Patch Link Table entries.

Table 1 contains a listing of the various statistics that we gathered for the three programs examined. The number of corrected software faults, shown in the first row, is the number of software defects that were corrected between the two versions of the software we examined. The next two rows show the total number of instructions in the release, and the total number of instructions involved in the software update. The area optimal number of patches is the number of patch regions that were created after spatially local patches were merged together. This number is greater than the number of bug fixes because often a single fix affects multiple regions of code. The next and final line in the table shows the average size, in number of instructions, of the optimally merged patch regions.

[2]The estimate we use for our results is conservative because the compiler tends to reschedule the code if anything has changed, and this rescheduling will be picked up as a change in the code by our patch generation software.

statistic	gs	zlib	sleepy
Software Faults Removed	26	3	9
Total Size of Release (in instrs)	103K	3.9K	18.2K
Total Size of Fixes (in instrs)	592	29	371
Total Size of Fixes (% of release)	0.57%	0.07%	2%
Area Optimal Number of Patches	33	6	16
Area Optimal Size per Patch (instr)	40.0	19.8	33.8

Table 1: Characterization of the bug fixes for the three different programs examined. Number of bug fixes is the number of software defects that were corrected between the two versions examined. Release size and fix size show the total number of instructions in the program and the total number of instructions involved in the software update. The area optimal number of patches is the number of patch regions that were created by the compiler after the patch merging step. This number is greater than the number of bug fixes because often a single fix affects multiple regions of code. After patch merging, the size per patch shows the average size of an individual patch region.

3.3 Software Engineering Models

We use the Non-Homogeneous Poisson Process (NHPP) Model of Goel and Okumoto [2] to model the rate at which defects in the software will be fixed. This is used to model the number of software faults left in a ROM when fabrication starts, which determines overall area and time saving of the PI-ROM in the next section. We use this model because of it's simplicity and applicability to a wide range of testing environments as noted by Misra [10] who used it to correctly predict the number of defects left in the space shuttle software subsystem.

Using this model, and some parameters that are defined from the engineering practices of the design team that would be using our system, we can estimate the number of defects left in the system versus time given the number of defects discovered so far and the timing between the discovery of those defects. We can also estimate the converse, the amount of time left until the system has only a certain number of defects left in it.

The Goel-Okumoto model is centered on two parameters, the total estimated number of defects in the system, b, and the parameter that reflects how quickly software faults can be found and fixed, denoted a. The fit for a can be found by examining several past chip designs and fitting the curve for varying values of b using least squares fit. b can be found for the current system by debugging for a given amount of time, and using the value previously calculated for a.

The function for the amount of defects left in the system after a time of test t is found as

$$m(t) = be^{-at}$$

The model works by fitting the data to an inverse exponential curve of the form presented in the equation above. This curve reflects the fact that a great deal of bugs can be found relatively quickly, but finding all of the bugs necessary to make the software suitably stable takes a great deal of

Description of Constant	Constant Value
Size of Code ROM	64K instructions
Bits per Instruction	16 bits
Initial Bugs per Line	20/1,000
Post-Test Bugs per Line	1/4,000
Instructions per Line	12
Patches perBug	2
Fabrication Time	60 days
Average Patch Size	45 instructions
a	0.05
ROM Density	1.0 by definition
FLASH Density	2.0 relative to ROM
Associative Density	52.0 relative to ROM

Table 2: The set of default constants used by our analytical model.

time. The intuition behind this is that more and more code must be examined to remove one more bug. The PI-ROM architecture takes advantage of this by having the majority of defects fixed during the early stages of test before fabrication begins. The remaining defects will take significantly longer to find and fix, and this will occur in parallel with the fabrication of the chip.

Using the Goel-Okumoto model of software reliability and the area model we can now make some calculations for various parameters effecting the model.

3.4 Area Versus Time Results

Given a set of parameters for the code size, engineering process, and desired reliability we can estimate how large of a patch system we will need and how long we will need to test in order to meet these constraints. We show the interaction by plotting the percent area increase over a simple ROM approach, versus the amount of time spent in testing and fabrication. The more time spent testing the less area that will need to be devoted to patching flaws in the system.

There are two cases to consider. In the first case, the time for fabrication is longer than the time required for testing, as may be the case for very small systems. In these systems the fabrication time is the bottleneck and the testing may be fully overlapped with fabrication time. In this case a good solution may be to simply use Flash to store all of the code because during the fabrication time we can complete all of the necessary testing. However, a better design point may still be to do some preliminary testing and build a PI-ROM, depending on the relative importance of cost and time-to-market.

The other case, which is more common, is where the testing time is longer than the fabrication time. Depending on the standing of the client, the technology used, and the demand for fabrication time, fabrication times can be between 50 and 100 days. Testing times depend on the software engineering practices of the programmers, the complexity of the tasks being performed in software, the number and training of the testing staff, and the size of the code.

As was discussed in section 3.3 we need to provide two inputs to our software engineering model, the parameter b, which is the number of bugs initially in the systems, and the parameter a, which captures the software engineering practices and the man power that is exerted on testing. There is also one more parameter needed to make the model complete, which is c – the amount of bugs that are remaining

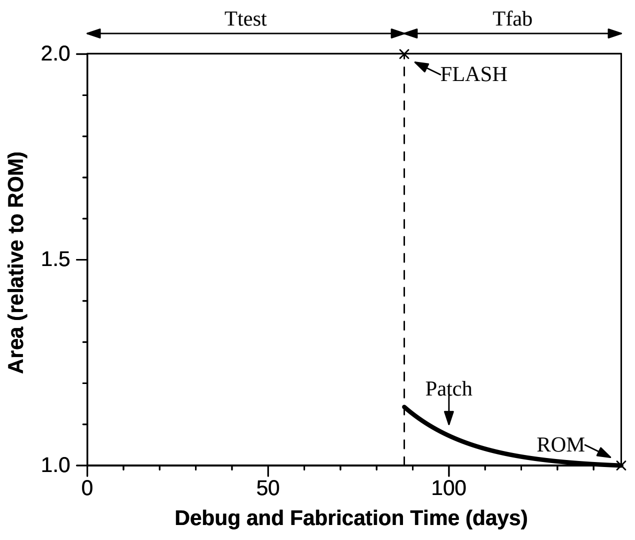

Figure 6: Area versus time for the three different memory architectures. The x-axis is the total amount of time for combined debug and fabrication measured in days, while the y-axis is the area of the system relative to the ROM. The time to market for the Flash is limited by the test time, which means that if we were to use a patch based architecture it is definitely beneficial to do some testing before we start fabrication. If we just use 25 days of test time before fabrication we can build a patch based architecture that is just 14.2% larger than ROM.

in the system when it is ready for release. Most embedded systems are never completely bug free, especially in the consumer market, and at some point the very small probability of the error occurring does not justify the ever increasing resources that must be applied to find it. We assume that if the code has less than c bugs in it, it is ready for release.

We start with how we get estimations of b and c. Software engineers have very effective rules of thumb for estimating how many bugs are in the system per line of code. Average code has 20 bugs for every 1,000 lines of code. Good code has 2 bugs for every 1,000 lines of code. We assume that at the start of debug we have "Average" code, and because of the embedded nature of our release we want the code quality to be "Excellent" with only 1 bug every 4,000 lines of code. There are other ways to measure the complexity of an application for the purposes of testing [3, 8] which involve measuring the number of operands and operators or control flow graphs of a program, but for the purpose of our evaluation the simple bugs per line model will suffice. From these estimates, and our knowledge of how many instructions correspond to a line of code and how big the final code is, we can estimate b and c. We then estimate a using reported testing times for various software code for embedded systems. We can now estimate area and total fabrication time using these parameters along with our area model.

The actual values of the constants that we used as our defaults are shown in Table 2. Figure 6 shows how area and total test and fabrication time interact for the three types of systems evaluated. The X-axis in figure 6 is the total amount of time for combined debug and fabrication measured in

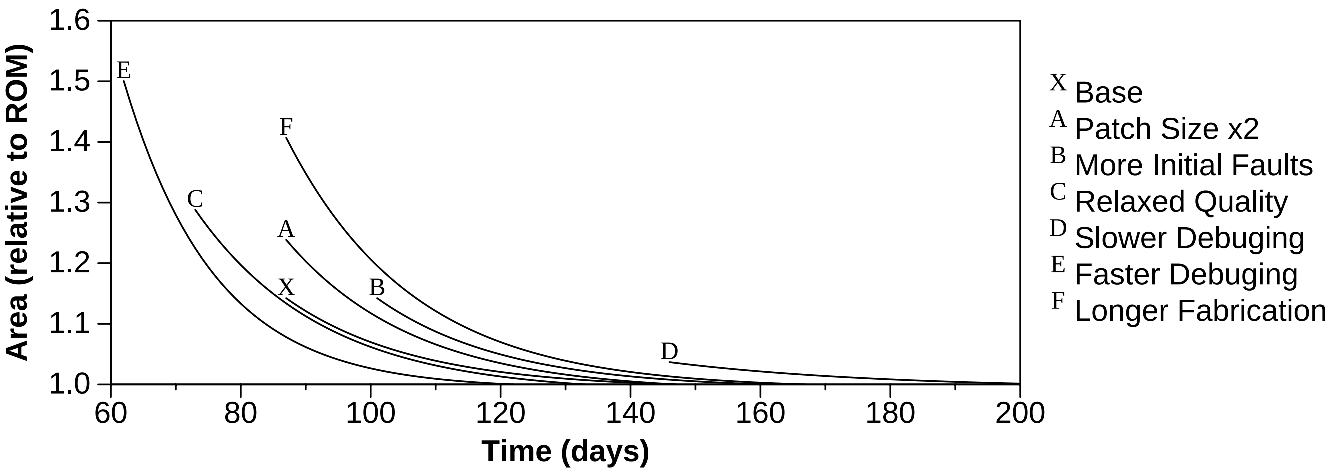

Figure 7: Area versus amount of debug time for a variety of different parameter sets. `Base` is the base configuration shown in Figure 6. `Patch Size x2` illustrates the area increase due to having patches of twice the normal size. `More Initial Faults` models the effect of having more bugs in the code to begin with. `Relaxed Quality` is what happens if the quality standards for the final product are lessened. `Faster Debugging` and `Slower Debugging` examine the effect of varying the parameter a from the Goel-Okumoto model and has the biggest effect on the performance of the system as a whole. The final curve, `Longer Fabrication`, shows what happens as we increase the time needed for fabrication relative to the total testing time needed.

days, while the Y-axis is the area of the instruction storage on the chip, as normalized to only using a ROM.

We first note the x-marks for ROM and Flash. The ROM is marked at an area of 1.0 with a total time of $T_{test} + T_{fab}$. Flash on the other hand takes up twice the area of the ROM but with the ability to get chips out limited only by the maximum of T_{test} and T_{fab}, which in this case is T_{test}.

The new option available is the PI-ROM, and it is shown on the graph with the solid dark line marked "Patch". All of the points on the curve are valid design options, and they show the tradeoff between time-to-market and area needed for software patching. As we wait longer and longer before beginning the fabrication of the chip, there will be less faults left in the software, and hence the area needed for any remaining patches will be less. This behavior can clearly be seen in figure 6 with the line for the PI-ROM starting out as high as 14% more area than a ROM, and converging to 0% as we reach $T_{test} + T_{fab}$ where we meet the design point for ROM.

Note that even if we take the same amount of time-to-market as a Flash based approach, the PI-ROM will only use 57% of the area that is used by Flash. This is because the minimum time is limited in this example by the testing time. Therefore, a given amount of testing can be done before we begin the fabrication of the chip, in effect giving us some testing time for free.

We now examine the tradeoffs from varying a variety of parameters in Figure 7. The line marked X is the same line as is presented in Figure 6, but the axis has been changed to show more detail. Just as in Figure 6 the first point on the curve is the first valid design point, at the minimum of T_{test} and T_{fab}. For each of the curves on the graph, the start of the line, where the letter is located, is the first point at which any design could be ready, and where a Flash based approach could be used at that time with an area of 2.0. The point where the curve reaches 1.0 for the Y-axis is the time where the purely ROM solution would occur for the given set of parameters. Each of the different curves on the graph is an examination of valid design points with a single parameter changed from the values used for X and shown in Table 2. Each curve starts at the min of T_{test} and T_{fab} for it's set of parameters and continues until it reaches a point equivalent to a ROM at $T_{test} + T_{fab}$.

The first curve, labeled A, shows the effect of having patches that are on average double the size of what was assumed in Table 2. Note that for the base case the size of the Patch Link Table and Patch Memory are quite close, so doubling the size of one structure does not add that much to the area overall. The results for B show what happens if we have a higher initial fault density (40 bugs per 1,000 lines of code). In this case, the area of the first valid design point does not increase over that of the base case, but instead we need to wait longer until we reach acceptable code quality before starting fabrication. As we increase the defect density, we increase the test time, but the amount of software defects discovered during the time for fabrication remains constant because the *rate* at which bugs are fixed does not change. For the curve marked C, the quality of the final product has been relaxed to (1 bug per 2,000 lines of code). In this case we can release sooner, so we get less time for testing before fabrication can start.

The curve D shows what happens if we change the value of a (to 0.03) in the Goel-Okumoto model to reflect a software testing methodology that takes more time to find and fix each bug. In this case T_{test} becomes much larger than T_{fab} and because of this there is a long time to do testing before it is desirable to start the fabrication. The tail of the model is exaggerated with slower debugging and the results show that we can achieve a very area efficient PI-ROM with the same time-to-market as a Flash based approach. The curve labeled E is perhaps the least intuitive of the results and is exactly the opposite of the effect shown for curve D. For

E, we assume that we debug the software at a very fast rate with a equal to 0.07. This results in the tail of testing being reduced significantly, and secondly the process has now become limited by T_{fab}. The final curve, F changes the time needed for fabrication to be almost equal to T_{test}.

4. RELATED WORK

In this section we discuss three areas of related research.

4.1 Patchable Control Store

The only prior work similar to ours that we are aware of is the Patchable Control Store used to fix code in ROM at boot time in the VAX-11/750. The VAX-11/750 has a purely ROM based control store of 6k x 80 bit words. Because there could sometimes be errors in the microcode stored in the ROM, the DEC engineers added a board which contained 1k x 80 bits of SRAM, PROMs to store the microcode updates, and 8k x 1bit of "flag" memory. At boot time the machine would copy the contents of the PROM into the SRAM, and initialize the flag memory. On every instruction access the flag bit is checked for that address. If the bit is high, then the instruction is retrieved from the ROM, otherwise it is retrieved from the SRAM that was initialized with the PROMs.

The implementation and design of the patchable control store differs from our work in the following respects. First, it requires the addition of area proportional to the size of the total instruction ROM, not proportional to the size of the patch area. For our approach, if there has been any amount of initial testing, the amount of patch code relative to the total code will be very small (as shown in Table 1). Adding area proportional to the total control store will significantly increase the cost of the chip. Secondly, the speed of the instruction lookup will be negatively impacted because the flag lookup is serialized with the instruction lookup for the VAX-11/750. The design we present has no additional latency when switching from fetching instructions in the ROM to the Patch Memory.

4.2 Incremental Compilation and Pro£ling

In recent years there have been several efforts to create compilers that can compile, profile, and optimize all incrementally. The target use of such technology is large software systems that take a very long time to both compile and profile. In these systems it is not efficient to have to recompile, relink, reprofile, and reoptimize the executable every time the code changes even slightly. Instead the idea is to be able to only update the sections of the code or profile that have actually changed in a significant way. These systems show that it is practical and feasible to update an optimized binary and detailed profiles after code changes due to defect fixes.

Even though these systems are not to be implemented in a non-updatable format such as a ROM, the problem is very similar – a significant amount of effort has been put into the original binary, and we now must change it while still using as much of the original effort as we possibly can.

Typically when a program is optimized for a system, it is first compiled into a state that is not fully optimized so that proper profiling can be done. This allows the profiled information to be tracked back to the source code that is responsible, unfortunately this also means that two different compilations must be done, one at a minimum level of optimization and one at a full level. In [1], Albert shows a way of mapping profile data from an optimized binary back to the source code. This sort of technique would be helpful for developing patch aware code generation.

The *Binary Matching Tool* (BMAT) system [12], takes a different approach to reducing development time. Instead of simply eliminating the extra compilation step, they propose to take older versions of the source code that have been compiled and profiled, and match them to the new version of the code. They do this using BMAT, which takes two versions of a binary and matches them together. This work is very similar to ours in that they are creating a map between two different versions of the same program, but differs in an important way. They are seeking *probabilisticly* good matches for the purpose of matching profile data, while we are interested in only exact matches for correctness.

4.3 Code Compression

A different approach to reducing code size in embedded systems is code compression, where the instructions are stored in a compressed form in memory and are decoded before their use.

LeFurgy et.al. [5] present an architecture for compressing code using dictionary-based compression. Their architecture makes use of a ROM that stores the compression instruction memory. The ROM can contain either an uncompressed representation of the memory or a codeword that is then converted to a dictionary address and length pointing to where the real instructions are stored. They show that by using this technique instruction ROMs may be compressed by 34% on average for the ARM instruction set.

LeFurgy and Mudge [6] show that these results are not limited to just general purpose embedded processors, but that the results also apply to other architectures such as DSPs. Lefurgy et. al [7] further show how these techniques may be implemented with more hardware efficiency using a software-based approach. There are many other compression techniques for embedded instruction memory, all of which can provide some reduction in the area of the ROM or RAM being used. All of these techniques are complimentary to our PI-ROM, and can be applied to our approach to provided additional savings in area. Code compression schemes, because it does not change the actual instructions, can be treated as a black box to our PI-ROM architecture in section 2.4.

5. SUMMARY

There are two commonly used approaches to storing instructions on a chip, either use a ROM or a Flash. ROM is highly optimized for density which minimizes the area per instruction, but because it can only be written at fabrication, it needs to be correct before the chip is taped out. This serializes the test of the embedded software with the actual fabrication of the chip. Another approach is to use a nonvolatile writable memory technology such as Flash. Because Flash can be written after the chip has been fabricated, the code testing and chip fabrication can be overlapped, thus reducing the total time to market. Unfortunately this comes at the price of increased area and hence cost. Instead, we present Patchable Instruction ROM (PI-ROM) as a technique to get density near that of a ROM with the limited post-fabrication writability to help decrease time to market.

PI-ROM works by allowing the system designer to fabricate a ROM even if it is not up to the required level of quality. The PI-ROM includes a hardware mechanism that works in concert with the compiler to patch the ROM on the fly in hardware. When the program's execution is about to enter a region of faulty code, the PI-ROM architecture intercepts the address and redirects the execution to a section of fixed code stored in the Patch Memory.

To examine the usefulness of the PI-ROM architecture, we first quantify how software patches effect the embedded programs at the architectural level. After examining the patch regions generated, we have found that there is a great deal of locality in how code is updated. We exploit this locality by merging minimum sized software updates into larger patch regions which can be implemented more efficiently. Merging non-contiguous patch regions that are within 16 instructions from one another is shown to be best.

Once we developed an understanding of the instruction level effects of code updates, we combined this information with an area model based on the bit densities of the three types of memory structures involved and models from software engineering. We have found that for reasonable values of fabrication time, software engineering methodologies, and chip sizes, using a PI-ROM can keep the design time afforded by using Flash, but use only 14% more area than a ROM. We further showed how these results were effected by changes in fabrication time, code quality, desired reliability, and average patch size. Even with changing these by a factor of two, we still are always left with excellent design points for using a PI-ROM that provide a good trade off between area and time to market.

The analysis we have done is pessimistic, and it is likely that real implemented systems will benefit even more from the use of Patchable Instruction ROM. The ROM we have used in all of these comparisons is based on a Masked-ROM where all the ROM values are programmed by the metal layers. If instead we were to use Diffused-ROM we would be able to even further reduce our area relative to Flash. In addition, we assumed a very aggressive debug rate. It is more likely that the debug rate will be closer to that presented for curve D in Figure 7. Taking all of this into consideration, Patchable Instruction ROM is a very attractive design option for instruction memory storage in embedded systems.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by DARPA/ITO under contract number DABT63-98-C-0045, by NSF CAREER grant No. CCR-9733278, and a grant from Compaq Computer Corporation.

6. REFERENCES

[1] G. Albert. A transparent method for correlating profiles with source programs. In *Second ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 1999.

[2] A. L. Goel. Software reliability models: Assumptions, limitations, and applicability. *Transactions on Software Engineering*, 12(11):1411–1423, 1985.

[3] M. H. Halstead. *Elements of Software Science.* Elsevier North-Holland, New York, 1977.

[4] H. Koike, F. Matsuoka, S. Hohkibara, E. Fukuda, K. Tomioka, H. Miyajima, K. Muraoka, N. Hayasaka, and M. Kimura. Quick-turnaround-time improvement for product development and transfer to mass production. *IEEE Transactions on Semiconductor Manufacturing*, 1(1), 1998.

[5] C. Lefurgy, P. Bird, I-C. Chen, and T. Mudge. Improving code density using compression techniques. In *30th International Symposium on Microarchitecture*, December 1997.

[6] C. Lefurgy and T. Mudge. Code compression for dsp. In *Workshop on Compiler and Architecture Support for Embedded Systems*, December 1998.

[7] C. Lefurgy, E. Piccininni, and T. Mudge. Reducing code size with run-time decompression. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, January 1998.

[8] T.J. McCabe and C.W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, December 1989.

[9] J. L. McWilliams, L. M. MacMillan, B. Pathak, and H. A. Talley. Ppa printer controller asic development. *Hewlett-Packard Journal*, 73(4):1–12, 1997.

[10] P. N. Misra. Software reliability analysis. *IBM Systems Journal*, 22(3):262–270, 1983.

[11] J. M. Mulder, N. T. Quach, and M. J. Flynn. An area model for on-chip memories and its application. *IEEE Journal of Solid-State Curcuits*, 26(2):98–106, 1991.

[12] K. Pierce and S. McFarling. Bmat – a binary matching tool. In *Second ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 1999.

[13] T. Sunaga. A 30-ns cycle time 4-mb mask rom. *IEEE Journal of Solid-State Curcuits*, 29(11):1353–1358, 1994.

[14] J. Tsouhlarakis, G. Vanhorebeek, G. Verhoeven, J.D. Blauwe, S. Kim, D. Wellenkens, P. Hendrickx, L. Haspeslagh, J.V. Houdt, and H. Maes. A flash memory technology with quasi-virtual ground array for low-cost embedded applications. *IEEE Journal of Solid-State Curcuits*, 36(6):969–978, 2001.