Agile Methods Applied to Embedded Firmware Development

Bill Greene
Intel Corporation
bill.greene@intel.com

Abstract

This paper describes the experience of applying Agile approaches to the development of firmware for the Intel® Itanium® processor family. Embedded development (i.e. firmware) projects are quite different from object-oriented and pure software endeavors, yet they face many of the same challenges that Agile software development practices address. Several unique challenges are described, including team members' specialized domain knowledge, technical backgrounds and attitudes toward change, and the impact hardware plays in firmware design. We found Agile approaches to be well-suited for our project, despite the fact that most Agile methodologists come from very different backgrounds.

1. Introduction

Embedded software development has a number of unique characteristics which distinguish it from application development. Fundamentally, embedded systems contain some amount of specialized hardware and firmware for a specific purpose. They are often deployed in unique environments which may place code size, real-time, and other performance requirements on the firmware. This often leads to the necessity of programming in low-level languages such as assembly language. In some cases the hardware is being developed along with the firmware, which requires the firmware to adapt to the quirks of the hardware. Many firmware developers are not classically trained software engineers, but often have electrical engineering backgrounds and pick up programming as necessary to solve the task at hand. Firmware developers also seem reluctant to adopt the latest software tools and methodologies, and they strive to develop code as expediently as possible.

Although most Agile software development literature cites its use for application development projects, often implemented in object-oriented languages, Agile techniques can be employed successfully on embedded projects because they share some common themes. In particular, Agile methodologies are targeted toward problems involving change and uncertainty, and are adaptive rather than predictive [1]. Firmware development can be very volatile, with changing requirements and hardware dependencies that force change without warning. Agile methodologies also emphasize collaboration and team interaction, valuing people over process. This can apply to any team endeavor, and is particularly suited to embedded development in which engineers are averse to oppressive processes. Last, Agile methodologies commonly advocate a barely sufficient process [2], which sits well with hardware engineers who aren't eager to try the latest process improvement edict from above.

2. Project Background

Our project is different from most typical examples of projects where Agile methodologies have been employed. We are developing firmware for a new family of Intel processors. This project has unique characteristics as well as commonalities with embedded development in other environments.

The Intel® Itanium® processor family [3] is a new 64-bit architecture developed to overcome performance limitations of traditional architectures and to provide performance headroom for the future. It supports 64-bit operating systems and provides binary compatibility with IA-32 applications.

The Itanium® architecture defines processor behavior that can be implemented completely in hardware, or with a combination of hardware and firmware. Our project, the Processor Abstraction Layer (PAL) firmware [4], provides a consistent firmware interface across all Itanium processor family designs to abstract processor implementation-specific features. Functionality includes processor initialization and self-test, run-time services to query and configure processor capabilities, error handling, power-management, new feature support, and other architecture features which can be done better, cheaper, or more flexibly in firmware. PAL also provides workarounds for processor errata, as it is almost always easier and cheaper to change firmware than to make hardware changes. It can take more than a month to get silicon changes back from the factory. PAL interacts with the processor through special mechanisms which provide back-door access to processor internals.

The PAL firmware consists of about 300,000 lines of Itanium® assembly code and 8,000 lines of C code. The limited availability of memory (to be used as a backing store for nested subroutine calls) during the early boot flow prevents us from coding more of the firmware in C.

3. Development environment

The PAL firmware team consists of seven developers and a manager/technical lead, and is part of the larger processor design team of well over a hundred hardware engineers. Two of the developers are located at a remote site and are currently focused on firmware support for a previous generation processor.

The hardware design team consists of microarchitects, logic designers, circuit designers, and mask designers who work for several years on Phase 1 of the design, which culminates in a manufactured processor.

In parallel with the hardware design team's Phase 1 efforts, the PAL firmware team is developing code to initialize and test the processor, and to implement error handling, testability, low-power, and other features. The hardware mechanisms and interfaces available for firmware's use are evolving as the hardware design proceeds. Change is continuous as tradeoffs are made to optimize performance and minimize silicon die area. Since firmware is much easier to change than the hardware, it is often asked to change to work with the evolving design. We strive to have full functionality in the firmware by the time silicon is available.

Phase 2 is the debug of the processor once silicon is available, and can take over one year to complete as several "steppings" or redesigns are made to fix functional issues and tune processor performance. In Phase 2, PAL firmware development becomes very reactive as fixes are required to workaround processor errata. At this point we are entirely driven by problems as they are discovered, and there is no visibility into upcoming requests for changes.

4. The need for change

Every project has challenges, and ours may not be much different than those found in other firmware or embedded designs. In fact, many of our problems seem similar to those which are cited by other teams looking at Agile methods.

Some of our most significant problems included:

- Frustration with detailed quarterly planning activities that resulted in schedules that couldn't be followed even one week after the planning session. People were feeling guilty that they couldn't follow their schedule, or they thought the planning process was a waste of time.
- Team members were too specialized in firmware domain knowledge, with very little cross-training (which is required to debug and find the root-cause of system problems once silicon is available). For example, we have different experts in processor error handling, initialization, and low-power features.
- Embarrassing testing escapes, due to a lack of test coverage. Changes were made without the safety net of a comprehensive regression suite in place, and sometimes without developing focused tests for those changes. We relied on other processor validation groups to find problems.
- Poor code maintainability, because the assembly code was sometimes overly optimized and complex. The code was littered with "kludges" and workarounds for errata in the hardware itself.
- Inconsistent coding style and coding techniques.

5. Agile development to the rescue

Our problems seemed to stem from the lack of an effective software development methodology, which is common in embedded projects. The majority of embedded software is developed by hardware engineers who learn just enough programming to get by. As embedded software projects increase in scope, the need for a better process is apparent, but it is not obvious where to find it.

I began exploring software engineering methodologies and practices for solutions to our problems. A friend at Raytheon Company told me about CMM and referred me to the NASA Software Engineering Institute publications [5]. While perhaps well-suited to large government projects, this was way too much process for our team to absorb and put into practice in a reasonable time.

I read parts of "Code Complete" [6], "Rapid Development" [7], and "Software Project Survival Guide" [8] by Steve McConnell. All of these are great books, but it would be difficult to bring all of these ideas to the team in such a way that they would be accepted. We needed something simple and tangible that would be embraced by the hardware engineers.

Stumbling upon a copy of "Extreme Programming Explained" [9], I finally found what we had been seeking – the values and practices of a successful software team that were easy to understand and seemed perfectly suited to address our team's problems.

I was convinced that the ideas in this book would address many of my concerns, would make the team more productive, and that we'd have more fun in the process. The key was collaboration and communication, and the book described simple practices that encourage this. I insisted that everyone in the group read the book, and everyone did except our most cynical team member (more about him later).

Shortly after the team read about Extreme Programming (XP), our group inherited a project from another firmware development team that had produced over 600 pages of requirements and implementation documentation but had very little working code when their project was canceled. What a timely way to drive home the value of working software over documentation!

6. Developers' concerns

Not everyone was excited as I was after reading the XP book. During initial discussion with the team, several serious concerns were raised:

- "How will I be evaluated at the end of the year?" Intel performance management is based on meritocracy, and ownership is very important in the rating process. Team members had always been encouraged to take on responsibility for important, visible projects and tasks. Collective code ownership seemed to undermine this, and XP's encouragement to do pair programming for all production code seemed to dilute the accomplishments of individual team members.
- "Pair programming – seems like a waste of time." But several veterans in the group had experience with "pair debugging" and found it to be very useful when troubleshooting system problems. We had experienced the synergy and quality which arises from collaborative effort in the debug environment.
- "With this emphasis on collaboration – would there be any privacy and time to do focused individual work?" While most engineers seem to enjoy working with others in moderation, we also look forward to time alone where we can think and create.
- Testing before coding seemed unusual to most of us, but we did see the value in this.

The key to the group's acceptance of XP was our decision to try several of the practices and see how they worked. If the team found them to be valuable, we would keep them and try others. If not, we would take what we learned and move on.

7. What we tried and the outcomes

In the end, we chose practices from Scrum [10] and XP and combined these ingredients into our own methodology which seemed to best meet our needs. From a management perspective, I found value in the Scrum methodology which provides a great organizational framework for some of the more developer-centric practices of XP.

Overall, adoption of these XP and Scrum practices was a gradual process. There was definitely the need for someone to champion and keep pushing this – changes in habits and behaviors don't come easily.

We are still experimenting, but I am convinced that methodology and process customization is good, and that being dogmatic about following *all* the practices in any Agile methodology is really not being *agile.*

7.1. Scrum

Scrum is a deceptively simple management and control process that manages the uncertainty in a project by planning work in 30-day intervals, called Sprints. The Sprint backlog, or tasks to be completed during the Sprint, is derived during the Sprint planning meeting from the product backlog, which enumerates all of the product features. Daily Scrums are held to keep the developers on track, and a Sprint retrospective serves to provide feedback to the process.

The following Scrum practices were of particular use to us:

7.1.1. Sprints (30-day iterations)

The 30-day Sprints provided a good granularity for planning, but some engineers didn't like the lack of visibility beyond 30 days. The team found this planning methodology useful because it allowed them to commit to completing well-understood tasks in a reasonable time frame while still allowing flexibility in responding to changes.

7.1.2. Sprint planning meeting

Emphasis on delivering business value (i.e. working features) in each iteration encourages completion of tasks: nothing is 90% done, it is either implemented and tested in the release or it is not. In the past we had a lot of partially done features for long periods of time, and it was hard to interpret how far along we really were.

7.1.3. Daily Scrum

This was highly effective at making team members aware of what others in the group are doing, and provides great feedback for the manager also. It also keeps the group focused on the right things; people don't get distracted for too long on side projects or stuck on blocking issues. However, team members grew tired of the daily meetings until we shifted the focus to "what are you working on today and what are the roadblocks?" and de-emphasized the question "what did you do yesterday?" The developers felt that the daily updates from team members were taking too much time and were not adding value to their work, so we kept this part brief.

7.1.4. Sprint review (retrospective)

These seemed very valuable at the beginning when lots of issues and questions were raised, but we found that they became shorter in duration as we completed more Sprints. The daily feedback and close interaction within the team has resulted in issues being addressed immediately, so we don't have as much to talk about at the retrospective.

7.2. Extreme Programming

Extreme programming is probably the best known Agile development methodology which focuses on the development process. We found the XP practices to be very effective at conveying software development best practices to engineers with hardware backgrounds.

The most valuable practice to our team was unit testing. The focus on testing improved everyone's motivation to invest more effort in test automation, tools, and infrastructure. This in turn made it easier to develop tests, and as more tests were developed people could see the quality increase, and the cycle fed upon itself.

The most controversial practice was pair programming, but it did encourage teamwork and collaboration. We found it useful once we fine-tuned the application to assembly language coding.

We found the practices of refactoring and collective ownership to be useful, but we struggled with general ownership issues. We realized that we still needed specialized domain knowledge, and this comes through ownership of technical areas. That ownership was then hard to let go when we had the opportunity for others to step in and help with some of the coding.

To some degree we were already making use of the practices of simple design, continuous integration, on-site customer, sustainable pace, and coding standards. XP's emphasis on these practices reinforced them for us.

We specifically didn't employ the practice of metaphor because we don't have an abstract problem to solve – there is a well-defined architecture which the firmware must implement, and this is well understood by the team.

7.2.1 Simple design

I have constantly reminded engineers that optimized assembly code is a nightmare to maintain, and that spending time to optimize code that is not performance critical is counterproductive. More collaboration due to pair programming and group ownership has made people more aware of their designs, and there has been some improvement in design quality.

7.2.2 Unit testing

The technique of writing tests first promoted an amazing shift in mind set. Tests were no longer an afterthought which may or may not find problems in completed code, but became a tool to support the coding itself. The tests provided an intermediate level specification, with more detail than the API defined by the Itanium® architecture. It forced us to think about what the code should be doing before coding, and made the coding process itself flow easily once the tests were in place.

On previous projects we had only developed functional tests which exercised the architected interfaces to the firmware. Unlike other languages for which test harnesses are widely available, there is no "AssemblyUnit" testing framework (that I know of). For our current project, we had to write our own tools which allowed us to run test setup code, branch to an arbitrary starting point in our firmware code, execute to an arbitrary ending point in our code, and then branch back to the test's checking code. With this capability it became easy to test any snippet of assembly code on the architectural simulator. The ability to code unit tests gave us the capability to test components at a low-level and then put them together into more complex structures with confidence. And the feelings of confidence reinforced the value of the testing.

Scripts were also developed to automatically run and check the results of our regression suite. The suite was broken into a 2-hour nightly regression of 1300 tests, and a less comprehensive 10-minute suite that is used by developers as they are coding. The ability to run a select group of tests with a simple command allows the tests to provide timely, useful feedback.

7.2.3. Refactoring

Refactoring is recoding or restructuring the system to improve the design without changing its behavior. This can be done by simplifying the design, improving the code's maintainability and/or consistency, adding flexibility, or making any other improvements in the low-level implementation. Now that we have a name for it, this activity occurs more often and people look for opportunities to improve the quality of existing code. This awareness had reduced the amount of "kludginess" introduced in coding, and is slowly improving the quality of legacy code.

7.2.4. Pair programming

We had used code inspections successfully in the past, and pair programming can be thought of as an extension of this process into the daily routine of developers. I think of pair programming as pulling the review process up from a discrete point in time to the instantaneous time at which the code is created. As code inspections find defects sooner and more efficiently than testing, pair programming finds defects sooner than inspections.

We found that pair programming in assembly language had some value during early stages of the design and coding, but that there came a point where it was not efficient. Instead of pair programming all production assembly code, we now advocate pairing during detailed design and initial coding, and then splitting up once the coding gets tedious. It's hard to quantify what this point is, but we left it up to the developers to decide when they had reached the point where having two people code assembly instructions loses its effectiveness. At that point the developers continue on their own, and then regroup to quickly review the code together.

Cross-training happened when pair programming was practiced, but not to the extent we had hoped. The reality is that the processor design is too complex for everyone to know – we need each person to have specialized domain knowledge (that is the engineers' real value). We can only hope to transfer some high-level knowledge as we strive to work together more.

Hard deadlines create problems for pair programming when one engineer has more domain knowledge. At crunch time, engineers feel that it is more expeditious for the expert to work on the implementation of some feature alone, and that pairing with another engineer who could be working on code with which he has expertise reduces group productivity. With five developers on site, we are trying a new approach where each person has a day of the week where they don't touch anything related to their core expertise. We're hoping this will reduce the tendency to return to one's expertise area for at least part of the week.

To addresses concerns about an individual's performance evaluation we emphasized a new evaluation criteria: teamwork. The openness of Agile allows the manager to see who is putting forth effort to help others. In my management role I needed to constantly remind engineers of the importance of the team effort – the success criteria is meeting team goals and how you go about doing that.

We tore down some cubicle walls, and most people like this better now. It literally lowers the barriers to communication and improves the closeness of the group. I noticed that the two developers who don't have direct eye contact with others in the team because of their cubicle orientation are not as involved with the rest of the group. It's almost as if the engineers with eye contact and who are located within earshot have a secret communication channel, where expressions of frustration elicit help from others who pick up on this.

We have two engineers who are located in different cities in another state, yet we were able to keep them involved through daily stand-up meetings with an audio bridge. We even successfully pair programmed on well-defined tasks, using Virtual Network Computing (VNC) for desktop sharing and telephone headsets for communication. In some ways this was even more convenient and engaging than pair programming at the same computer.

Interestingly enough, our biggest initial skeptic now does the most pair programming in the team! This person was motivated by group discussion and seeing first hand how XP practices worked, rather than by being forced to read the XP book.

7.2.5. Collective ownership

Ownership is hard to give up. People always gravitate to the area they have domain expertise in, and they feel responsible for progress in this area. However, engineers have accepted that others may improve their code and don't get upset when it happens.

7.2.6. Continuous integration

This was never a problem for us, as the firmware architecture is very modular and allows development to proceed in parallel in several areas. CVS forces people to integrate changes from the repository into their private workspaces before they can commit changes, so integration is done every couple of days or so with no bad side effects.

7.2.7. On-site customer

Since the PAL firmware architecture is already defined, there is no external customer to negotiate with. In my role as the technical lead, I was able to make recommendations for prioritization of deliverable functionality as a customer proxy. However, the hardware design team is frequently requesting firmware changes to support the processor design, and our team is collocated with and closely integrated organizationally with the hardware design team. Sitting among the design team promoted good interaction and early discussion of alternatives. If we had been physically separated, it is more likely that we would have been given more formal demands for requirements that would have been harder to negotiate.

7.2.8. Sustainable pace

While we agree with this in principle, hard deadlines created the need to work extra hours at times. Certain features are needed to enable the whole processor design project and we are often in the critical path. We strive for balance in the longer term.

7.2.9. Coding standards

Assembly language by its nature can be quite unwieldy if no standards are in place to keep it organized. Several years ago we developed coding conventions for Itanium® assembly, and they have kept our code somewhat sane. The real problem occurs when people hack a workaround into unfamiliar code, and then do not feel comfortable refactoring due to the lack of a complete regression suite in place. We hope to remedy this on our project by having a regression suite that encourages people to refactor as they are coding workarounds and changes after silicon is available.

8. Observations

Hardware design is in constant flux as tradeoffs are made for silicon area, performance, and functionality; especially in the back-door mechanisms that PAL firmware interfaces with. The attitude of "embracing change" [9] that XP and other Agile methodologies promote is perfect to accommodate this.

In our project, external product requirements (i.e. the PAL firmware architecture) are rigorously defined but the hardware interface available to us changes. The challenge is to bridge the gap between the architecture and processor hardware implementation. We don't need to negotiate with an external customer on features, but we do need to negotiate with hardware designers on tradeoffs between hardware and firmware to meet design requirements. An analogy for non-embedded software developers is that the customer's requirements are well-defined, but the programming language is changing!

A key difference between our embedded design and pure software development is that in our project, processor design domain knowledge is the key skill, not programming. Our efforts to cross-train throughout the group were only partially successful because of this. We will continue pair programming where possible to increase knowledge sharing and code quality, but we must acknowledge that the individual expertise in the group is actually a valuable asset.

It was noted by the manager and technical leads of one of our peer groups that there was no externally visible adverse impact as a result of our adopting an Agile methodology. In fact, they actually noticed that our team was more approachable and willing to respond to change requests. During external interaction with other teams, individuals who have specific domain knowledge are still the primary contacts for these areas. Even though the development may be conducted through pair programming, having a single point of contact for areas of domain expertise provides a level of comfort to other teams.

The emphasis of Agile methodologies on testing is extremely valuable, and this brought about the biggest change to the bottom line: the quality of our code. Unfortunately you can never really be done testing, but the practice of "test first" puts the proper focus on testing in the minds of developers. Our addition of unit tests has doubled the size of our test base over previous projects at the same point in development. We'll know when silicon is available and our firmware is delivered to customers how we've done, but we're confident that our defect rate will be dramatically lower than on past projects.

A downside to the increased collaboration that Agile methodologies foster and encourage is that it exposes people to each other to a greater extent than some are used to or comfortable with. One individual in our team was frustrated with having to work so closely with others during a period where we were insisting on pair programming, yet another engineer eventually left the group because he didn't feel we were acting *enough* as a team.

Through all this I have learned that reading about methodologies and team interaction is easy, but when you are dealing with real people it becomes *much* harder. Despite the challenge that Agile brings with the close interaction of the team, I think the benefits far outweigh the challenges. Agile methodologies aren't a panacea, but they go a long way toward improving the development process.

9. Conclusion

Our experiences with Agile methodologies have been mostly positive, but we found that full adoption is a slow process. Since we are now employing most of the Scrum and XP practices that we feel are applicable to firmware development, going forward we will be focusing on improving our execution of these practices. In particular, there is always room for improvement in the "people" aspect of Agile. We hope to improve our collaboration and communication which lies at the heart of Agile methodologies.

Other project managers within the processor design team have shown interest in what we are doing. They've heard of Agile and are intrigued by it, but they don't know much about it. Since processor design can be viewed as a big software project written in a hardware description language (e.g. Verilog), perhaps Agile methodologies could be applied here as well. Since the heritage of Agile comes from other disciplines such as manufacturing and process development, there is no reason to think it could not. In fact, I have noticed that some processor design teams within Intel are operating in a more agile fashion, and they seem to be more successful.

10. About the author

Bill Greene is the manager and technical lead for Itanium® Processor Family firmware development at Intel Corporation. He joined Intel in 1994 and worked on the development of the first-generation Itanium® processor, then transitioned to firmware development for the Itanium®, Itanium® 2, and future Itanium® Processor Family processors. Bill holds Bachelors and Masters Degrees in Computer and Electrical Engineering from Purdue University and is a Certified Scrum Master. He can be reached at bill.greene@intel.com.

10. References

[1] Fowler, M., *The New Methodology*, http://martinefowler.com

[2] Cockburn, A., *Agile Software Development,* Addison-Wesley, 2002.

[3] Intel® Itanium® Processor Family web site, http://developer.intel.com/design/itanium/family/

[4] *Intel® Itanium® Architecture Software Developer's Manual,* Intel Corporation, 2003, vol. 2, ch. 11. http://developer.intel.com/design/itanium/manuals/iiasdmanual.htm

[5] *Recommended Approach to Software Development,* NASA SEL, 1992.

[6] McConnell, S., *Code Complete,* Microsoft Press, 1993.

[7] McConnell, S., *Rapid Development,* Microsoft Press, 1996.

[8] McConnell, S., *Software Project Survival Guide,* Microsoft Press, 1997.

[9] Beck, K., *Extreme Programming Explained,* Addison-Wesley, 2000.

[10] Schwaber, K., *Agile Software Development with Scrum*, Prentice Hall, 2002.