

Frontier Journal

Frontier Visionary Interview

[Prof. Robert J. Aumann](#), Hebrew Univ. of Jerusalem, Nobel Laureate in Economics

[Prof. A. Michael Spence](#), Stanford Univ., Nobel Laureate in Economics

[Prof. Martin L. Perl](#), Stanford Univ., Nobel Laureate in Physics

[Prof. Frank Wilczek](#), MIT, Nobel Laureate in Physics

[Steve Wozniak](#), Co-founder, Apple Computer

[Vinton G Cerf](#), Turing Award Winner

[Ann Winblad](#), Co-founder, Hummer Winblad Venture Partners

[Richard Stallman](#), Founder of GNU Project

[Jim Rogers](#), American Investor

[Alan Kay](#), PhD, Turing Award Winner

[Prof. Bjarne Stroustrup](#), Man behind C++, Texas A & M Univ.

[Brian Behlendorf](#), Co-founder of Apache Project

[Rajeev Madhavan](#), Co-founder, Chairman & CEO, Magma Design Automation

Jimmy Wales, Founder of Wikipedia

[Craig Newmark](#), Founder of Craigslist.org

[Greg Gianforte](#), Founder & CEO of RightNow Technologies, Inc

[Grady Booch](#), Chief Scientist, IBM Rational

[Aart de Geus](#), PhD, Co-founder, Chairman & CEO, Synopsys

Content

| | |
|---|----|
| Did Brooks Envision the Agile Approach 30 Years Ago? <i>Orit Hazzan</i> | 3 |
| A Compacting Real-Time Memory Management System <i>Silviu S. Craciunas et al.</i> | 16 |
| Decoupling dynamic program analysis from execution in virtual environments <i>Jim Chow et al.</i> | 39 |
| Bridging the Gap between Software and Hardware Techniques for I/O Virtualization <i>Jose Renato Santos et al.</i> | 66 |

Did Brooks Envision the Agile Approach 30 Years Ago?

Orit Hazzan

Department of Education in Technology and Science

Technion - Israel Institute of Technology

Haifa 32000, Israel

Email: oritha@technion.ac.il

Abstract

This essay analyzes the classic book *The Mythical Man-Month* from the perspective of agile software development. It illustrates how, some thirty years ago, in 1975, Brooks, the author of *The Mythical Man-Month*, discussed some of the basic ideas of agile or software development that started receiving more and more attention from software practitioners and more and more software development. This book was written at a time when the nature of the field of software engineering and the nature of software systems developed at that time were completely different than they were ten years ago, when the agile approach of software houses adopted it as their primary development process. The analysis presented in this essay provides the community of software engineering professionals with a historical perspective on software system development processes.

1. Introduction

In your opinion, when was the following quote written?

Milestones or Millstones?

How does one control a big project on a tight schedule? The first step is to have a schedule. Each of a list of events, called milestones, has a date. Picking the dates is an estimating problem [...] crucially dependent on experience.

For picking the millstones there is only one relevant rule. Millstones must be concrete, specific, measurable events, defined with knife-edge sharpness. Coding, for a counterexample, is “90 percent finished” for half of the total coding time. Debugging is “99 percent complete” most of the time. “Planning complete” is an event one can proclaim almost as will.

Concrete milestones, on the other hand, are 100-percent events. “Specifications signed by architects and implementers,” “source coding 100 percent complete...,” “debugged version passes all test cases.” These concrete milestones demark the vague phases of planning, coding, debugging. (Brooks, 1975, p. 154)

This quote clearly reflects one of the main ideas of the agile paradigm for software development. It is therefore reasonable to assume that it was written at some point during the past decade, when the agile approach became predominant and was adopted by many software houses. However, this quote was written by Brooks 33 years

ago, in 1975, in his classic book The Mythical Man-Month.

Frederick P. Brooks is one of the fathers of software engineering (SE). His famous book, The Mythical-Man Month, is one of the classic manuscripts of the SE community and, as illustrated in this essay, remains relevant till today, over thirty years after it was written. Brooks' essay No silver bullet: Essence and accidents of software engineering (1987) is considered to be another milestone of SE.

The above quote is only one illustrative example of the main message of this essay, in which I attempt to deliver the idea that many fundamental ideas of agile software development were already presented by Brooks over thirty years ago. For this purpose, I classify quotes taken from Brooks' The Mythical Man-Month manuscript according to the four principles of the Agile Manifesto (see Table 1). The Agile Manifesto was formulated in 2001 by a group of 17 leading software practitioners and its principles are accepted by the community of agile development, regardless of the specific agile method used.

Table 1. The Agile Manifesto (<http://agilemanifesto.org/>)

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it.

Through this work we have come to value:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

While reading this essay, the readership readers are asked to recall the nature of the period during which Brooks wrote his manuscript.

First, Brooks formulated his ideas before the e-communication era, when even elementary e-communication tools were not widely available and used. This fact may explain, for example, Brooks' high awareness to human communication. This era is also reflected in Brooks' writing style, for example, by not using gender-balanced terminology.

Second, Brooks laid out his ideas mainly based on his experience with the development process of the OS/360 operating system; today, as is known, common software projects do not deal with operating system development.

Third, when Brooks wrote *The Mythical Man-Month*, most software development processes were waterfall-model based, even though this model was known to be an ineffective and inappropriate model for software development processes. The dominance of this development process, however, is explained by Brooks in his essay *No silver bullet: Essence and accidents of software engineering manuscript* (Brooks, 1987):

The waterfall model, which was the way most people thought about software projects in 1975, unfortunately got enshrined into DOD-STD-2167, the Department of Defense specification for all military software. This ensured that its survival well past the time when most thoughtful practitioners had recognized its inadequacy and abandoned it. Fortunately, the DoD has since begun to see the light. (p. 266)

This essay is organized as follows. Section 2 presents quotes from Brooks' book according to the four principles of the Agile Manifesto. The objective of this presentation style, as mentioned above, is to show that Brooks introduced ideas that three decades later, became the fundamentals of the agile approach for software development. In Section 3, I continue to illustrate the similarity of Brooks' ideas to the agile approach by presenting Brooks' perspective on software development processes, twelve and twenty years after the first edition of *The Mythical Man-Month* was published. This is done by quoting relevant excerpts from *No silver bullet: Essence and accidents of software engineering* (1987) and from the chapter Brooks added to the second addition of *The Mythical Man Month* published in 1995. In Section 4, I conclude.

I would like to make two remarks about the writing style of this essay. First, since most of the quotes in this

essay are taken from the second addition of *The Mythical Man Month* (1995), only page numbers are given for each quote. If the quote is taken from another resource, it is specified in full. Second, additional quotes, beyond the ones presented in this essay, could have been selected for presentation in this essay in order to deliver its message; however, due to space limitations, I chose the quotes that seem to me to be most illustrative.

2. The Agile Approach from Brooks' Perspective in 1975

In this section, I present quotes from Brooks' *The Mythical Man-Month* book and arrange them according to the four principles of the Agile Manifesto . This presentation aims to illustrate the similarities between Brooks' approach and the agile approach with respect to the management of software projects.

2.1. Individuals and interactions over processes and tools

Brooks explains the importance of individuals and their interactions by the dependencies that exist among developers:

The dependence upon others has a particular case that is especially painful for the system programmer. He depends upon other people's programs. These are often maldesigned, poorly implemented, incompletely delivered (no source code or test cases), and poorly documented. So he must spend hours studying and fixing things that in an ideal world would be complete, available, and usable. (p. 8)

The source of Brooks' famous law - Adding manpower to a late project makes it later (p. 25) - is also explained by the communication among team members that is required in software development processes:

Since software construction is inherently a systems efforts—an exercise in complex interrelationships—communication effort is great, and it quickly dominates the decrease in individual task time brought about by partitioning. Adding more men then lengthens, not shortens, the schedule. (p. 19)

Further, Brooks suggests that the rationale for his law results from the fact that most of the cost is spent on communication between people:

I have earlier argued that the sheer number of minds to be coordinated affects the cost of the effort, for a major part of the cost is communication and correcting the ill effects of miscommunication (system debugging). (p. 30)

One means by which agile software development fosters communication is daily and short stand-up meetings, in

which each team member reports on the progress he or she made the previous day, what he or she is going to accomplish today and, optionally, what problems he or she is facing. These stand-up meetings are very short and last no longer than ten minutes; yet, they enhance communication and increase the transparency of the development process. Brooks suggests a similar idea in reference to weekly meetings:

The fruitfulness of these [weekly] meetings springs from several sources:

- 1.The same group – architects, users, and implementers – meets weekly for months. No time is needed for bringing people up to date.
- 2.The group is bright, resourceful, well versed in the issue, and deeply involved in the outcome. No one has an “advisory” role. Everyone is authorized to make binding commitments.
- 3.When problems are raised, solutions are sought both within and outside the obvious boundaries.
- 4.The formality of written proposals focuses attention, forces decision, and avoids committee-drafted inconsistencies.
- 5.The clear vesting of decision-making power in the chief architect avoids compromise and delay. (p. 67)

The consequences of this on-going communication are clear:

These “fall festivals” were useful not only for resolving decisions, but also for getting them accepted. Everyone was heard, everyone participated, everyone understood better the intricate constraints and interrelationships among decisions. (p. 68)

As an illustrative example of the importance of communication, Brooks analyzes the case of the tower of Babel presented in the bible and applies the lessons of historical events for software projects. In the spirit of the first principle of the Agile Manifesto, Brooks suggests enhancing communication among the team members by using the available communication tools at that time:

How, then, shall teams communicate with one another? In as many ways as possible:

- 1.Informally. Good telephone service and a clear definition of intergroup dependencies will encourage the hundreds of calls upon which common interpretation of written documents depends.

2.Meetings. Regular project meetings, with one team after another giving technical briefings, are invaluable. Hundreds of minor misunderstandings get smoked out this way.

3.Workbook. A formal project workbook must be started at the beginning, This deserves a section by itself. (p. 75)

With respect to the Workbook, I would like to mention the Tracker role in agile teams, whose role is to monitor the team' s work. This role includes (a) measuring the group' s progress using measures defined by the team, the customer, and the organization and (b) arranging the collaborative workspace boards and the team diary/collective memory.

I conclude the illustration of the first principle of the Agile Manifesto with a quote that shows the importance Brooks attributed to communication, by not restricting it to the team members, but rather, by asserting its importance for the managerial level as well:

... only a small part—perhaps 20 percent—of the executive' s time is spent on tasks where he needs information from outside his head. The rest is communication: hearing, reporting, teaching, exhorting, counseling, encouraging. (p. 112)

2.2. Working software over comprehensive documentation

Under this principle of the Agile Manifesto, I present Brooks' quotes which express concerns related to quality practices such as testing. The following excerpt is an illustrative quote:

So testing drags on and on, the last difficult bugs taking more time to find than the first. (p. 9)

Because of optimism, we usually expect the number of bugs to be smaller than it turns out to be. Therefore testing is usually the most mis-scheduled part of programming. (p. 19-20)

Due to the recognized difficulties in estimating time for the testing activity, and in a spirit similar to that of agile software development, Brooks hints at the need to start testing early and throughout the entire development process:

Time after time, the careful product tester will find places where the word didn' t get passed, where the design

decisions were not properly understood or accurately implemented. For this reason such a testing group is a necessary link in the chain by which the design word is passed, a link that needs to operate early and simultaneously with design. (p. 69)

Furthermore, Brooks advocates the need for a systematic testing process:

The unexpectedly hard part of building a programming system is system test. I have already discussed some of the reasons for both the difficulty and its unexpectedness. From all of that, one should be convinced of two things: system debugging will take longer than one expects, and its difficulty justifies a thoroughly systematic and planned approach. (p. 147)

As with a test-driven development process (Beck, 2002), the development process may end with more test lines than code lines that the test tests:

Build plenty of scaffolding. By scaffolding I mean all programs and data built for debugging purposes but never intended to be in the final product. It is not unreasonable that for there to be half as much code in scaffolding as there is in product. (p. 148)

And, similar to Beck (2000) who explains the need for a special testing process, by stating Work with human nature, not against it (p. 116), Brooks says:

Add one component at a time. This project, too, is obvious, but optimism and laziness tempt us to violate it. To do it requires dummies and other scaffolding, and that takes work. And after all, perhaps all that work won't be needed? Perhaps there are no bugs?

No! Resist the temptation! That is what systematic testing is all about. One must assume that there will be lots of bugs, and plan an orderly procedure for snacking them out. (p. 149-150)

Brooks refers also to regression tests, which in agile software development is integrated with the practice of continuous integration:

As a consequence of the introduction of new bugs, program maintenance requires far more system testing per statement written than any other programming. Theoretically, after each fix one must run the entire bank of test

cases previously run against the system, to ensure that it has not been damaged in an obscure way. In practice such regression testing must indeed approximate this theoretical ideal, and it is very costly. (p. 122)

Team velocity, another concept used in agile software development processes, can be interpreted as the amount of productive work units per iteration (Beck and Fowler, 2000). Measuring team velocity per iteration increases visibility of the development progress and enables to make decisions with respect to functionalities and priorities. The following quote reflects, in Brooks' terminology, the need for measuring the team velocity:

The estimates were very careful, done by experienced teams estimating man-hours for several hundreds subtasks on a PERT chart. When the slippage pattern appeared, he asked them to keep careful daily logs of time usage. These showed that the estimating error could be entirely accounted for by the fact that his team were only realizing 50 percent of the working week as actual programming and debugging time. Machine downtime, higher-priority short unrelated jobs, meetings, paperwork, company business, sickness, personal time, etc. accounted for the rest. In short, the estimates made an unrealistic assumption about the number of technical work hours per man-year. (p. 89-90)

The last agile idea I introduce, in Brooks' terminology, under the second principle of the Agile Manifesto is refactoring (Fowler, 1999), which implies that agile software development legitimizes time allocation for code improvement in order to enhance code readability and understanding:

In most projects, the first system built is barely usable. It may be too slow, too big, awkward to use, or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved. The discard and redesign may be done in one lump, or it may be done piece-by-piece. But all large-system experience shows that it will be done. Where a new system concept or new technology is used, one has to build a system to throw away, for even the best planning is not so omniscient as to get it right the first time. (p. 116)

I end this section by referring to the right side of the second principle of the Agile Manifesto - Working software over comprehensive documentation - and I examine Brooks' perspective of documentation. In the spirit of this principle, Brooks agrees that documentation is needed, though it should satisfy some requirements, such as being on a high level of abstraction.

Every user needs a prose description of the program. Most documentation fails in giving too little overview. The trees are described, the bark and leaves are commented, but there is no map of the forest. To write a useful prose description, stand way back and come in slowly [...].

For the modifier, as well as the more casual user, the crying need is for a clear, sharp overview... (p. 165-166)

The need for such documentation is derived from the following objective, which clearly reflects the spirit of the second principle of the Agile Manifesto:

As a principal objective, we must attempt to minimize the burden of documentation, the burden neither we nor our predecessors have been able to bear successfully. (p. 172)

2.3 Customer collaboration over contract negotiation

This principle of the Agile Manifesto is reflected in Brooks' acknowledgments of the importance of establishing transparent and trustful relationships with the customers:

The management question, therefore, is not whether to build a pilot system and to throw it away. You will do that. The only question is whether to plan in advance to build a throwaway, or to promise to deliver the throwaway to customers. Seen this way, the answer is much clearer. Delivering the throwaway to customers buys time, but it does so only at the cost of agony for the user, distraction from the builders while they do the redesign, and a bad reputation for the product that the best redesign will find hard to live down. (p. 116)

Unlike the other three principles of the Agile Manifesto, I did not find many illustrative quotes in Brooks' book that correspond to this principle. This can be explained by the reference made in the introduction to the essay to the era in which Brooks wrote his book and the kind of software projects that formed the basis of Brooks' writings. Specifically, since The Mythical Man-Month relies mainly on the building process of an operating system, direct exposure to customers and users was not a regular event as it is in many of the software projects developed in the past decade.

2.4. Responding to change over following a plan

This principle of the Agile Manifesto is clearly reflected in Brooks' following statement:

The Only Constancy Is Change Itself

Once one recognizes that a pilot system must be built and discarded, and that a redesign with changed ideas is inevitable, it becomes useful to face the whole phenomenon of change. The first step is to accept the fact of change as a way of life, rather than an untoward and annoying exception...

Both the tractability and the invisibility of the software product expose its builders to perpetual changes in requirements.

... some changes in objectives are inevitable, and it is better to be prepared for them than to assume that they won't come. Not only are changes in objective investable, changes in development strategy and techniques are also investable. The throw-one-away concept is itself just an acceptance of the fact that as one learns, he changes the design. (p. 117)

Since the message of this quote is so straight forward, I find it sufficient to illustrate how Brooks viewed this principle of the Agile Manifesto. To further highlight the agile nature of this claim, it is sufficient to compare its message to Beck's phrase Embrace Change, which is part of the title of the first edition of his book, Extreme Programming Explained: Embrace Change (Beck, 2000).

3. Brooks' Perspective after Twelve (1987) and Twenty (1995) Years

This section shows how Brooks continued to introduce ideas in the spirit of agile software development in two of his future writings. The first is No Silver Bullet - Essence and Accident in Software Engineering published in 1987, which is presented as Chapter 16 in the 2nd edition of The Mythical Man-Month published in 1995. The second is Chapter 19 of the 2nd edition of The Mythical Man-Month entitled The Mythical Man-Month after 20 Years¹. Although presented here more briefly, Brooks' agile-related ideas from 1987 and 1995 are also organized according to the four principles of the Agile Manifesto. When reading the following quotes, the readers are asked to remember that between 1975, when the first edition of The Mythical Man-Month was published, and the time in which these two later manuscripts were published, the nature of software systems changed significantly.

3.1. After twelve years: No Silver Bullets - Essence and Accident in Software Engineering (Chapter 16 in the 2nd edition of The Mythical Man-Month, 1995)

As mentioned, in 1987, twelve years after the 1st edition of The Mythical Man-Month was published, Brooks

wrote another classic SE manuscript - No Silver Bullet - Essence and Accident in Software Engineering, which was added to the 2nd edition of The Mythical Man-Month, published on 1995, as Chapter 16. In this manuscript, Brooks naturally included new ideas, some of which are, once again, agile-oriented, as illustrated according to the four principles of the Agile Manifesto.

Individuals and interactions over processes and tools

To illustrate this principle of the Agile Manifesto, I chose one of Brooks' quotes that highlights the attention Brooks gave to social and cognitive aspects of software development processes, thus focusing on the individuals and their interactions.

In spite of progress in restricting and simplifying the structure of software, they remain inherently unvisualizable, thus depriving the mind of some of its most powerful conceptual tools. This lack not only impedes the process of design within one mind, it severely hinders communication among minds. (p. 186)

Working software over comprehensive documentation

To illustrate this principle of the Agile Manifesto, I present the idea of a gradual software construction process in which the software is run at each stage. Such a construction process is similar to development processes inspired by test-driven development, in which the entire system is testable at each stage.

Some years ago Harlan Mills proposed that any software system should be grown by incremental development. That is, the system should first be made to run, even though it does nothing useful expect call the proper set of dummy subprograms. Then, bit by bit it is fleshed out, with the subprograms in turn being developed into actions and calls to empty stubs in the level below.

In the continuation of this quote, Brooks' mentions the impact of such a development process on the team's morale, and thus, this quote also supports the first principle of the Agile Manifesto, which conveys the importance of the individual in development teams.

The morale effects are startling. Enthusiasm jumps when there is a running system, even a simple one. Efforts redouble when the first picture from a new graphics software system appears on the screen, even if it only a rectangle. One always has, at every stage of the process, a working system. I find that teams can grow much

more complex entities in four months than they can build. (p. 201)

Customer collaboration over contract negotiation

The following quote states this principle of the Agile Manifesto very clearly. One aspect of this principle is that clients keep improving their understanding of their desired system throughout the development process, and therefore, a software development process should help clients gradually grasp their needs and construct their understanding of the developed system in a gradual process. Agile software development achieves this by applying several practices, such as short releases and planning sessions in which customers participate. The following quote illustrates this idea as Brooks voiced it:

Therefore the most important function that software builders do for their clients is the iterative extraction and refinement of the product requirements. For the truth is, the clients do not know what they want. They usually do not know what questions must be answered, and they almost never have thought of the problem in detail that must be specified. Even the simple answer—"Make the new software system work like our old manual information-processing system"—is in fact too simple. Clients never want exactly that. Complex software systems are, moreover, things that act, that move, that work. The dynamics of that action are hard to imagine. So in planning any software activity, it is necessary to allow for an extensive iteration between the client and then designer as part of the system definition. (p. 199)

This message is further emphasized in the following quote:

I would go a step further and assert that it is really impossible for clients, even those working with software engineers, to specify completely, precisely, and correctly the exact requirements of a modern software product before having built and tried some versions of the product they are specifying. (p. 200)

Responding to change over following a plan

This principle of the Agile Manifesto is reflected by the following quote by Brooks, which calls for a different approach towards software development processes that enables the introduction of change into the development process.

Much of present-day software acquisition procedures rests upon the assumption that one can specify a satisfactory

system in advance, get bids for its construction, have it built, and install it. I think this assumption is fundamentally wrong, and that many software acquisition problems spring from that fallacy. Hence they cannot be fixed without fundamental revision, one that provides for iterative development and specialization of prototypes and products. (p. 200)

3.2. After twenty years: The Mythical Man-Month after 20 Years (Chapter 19 in the 2nd edition of The Mythical Man-Month, 1995)

Chapter 19 of the 2nd edition of The Mythical Man-Month - The Mythical Man-Month after 20 Years - also hints at what later became agile software development. This assertion is illustrated in what follows, by addressing the four principles of the Agile Manifesto.

Individuals and interactions over processes and tools

The title of the following quote, as well as the quote itself, clearly reflects the spirit of this principle of the Agile Manifesto. Furthermore, it advocates the importance of research on people-related issues in SE.

People Are Everything (Well, Almost Everything)

Some readers have found it curious that The MM-M devotes most of the essays to the managerial aspects of software engineering, rather than the many technical issues. This bias was due in part to the nature of my role on the IBM Operating System/360 (now MVS/370). More fundamentally, it sprang from a conviction that the quality of the people of the project, and their organization and management, are much more important factors in success than are the tools they use or the technical approaches they take.

...

Most academic research on software engineering has concentrated on tools. I admire and covet sharp tools. Nevertheless, it is encouraging to see ongoing research efforts on the care, growing, and feeding of people, and on the dynamics of software management. (p. 276)

Working software over comprehensive documentation

This principle of the Agile Manifesto is partially reflected in the following quote, in which Brooks outlines one of

the problems with the waterfall model. Brooks emphasizes the complexity of testing processes, pointing to the pitfalls of the waterfall model that ignores this complexity. He continues with a suggestion of how to obtain working software by presenting an agile-oriented process of software development that is based on elements such as short iterations, testing, and schedule.

The second fallacy of the waterfall model is that it assumes one builds a whole system at once, combining the pieces for an end-to-end system test after all of the implementation design, most of the coding and much of the component testing has been done. (p. 266)

Harlan Mills ... early advocated that we should build the basic polling loop of a real time system, with subroutines calls (stubs) for all the functions ... but only null subroutines. Compile it; test it. It goes round and round, doing literally nothing, but doing it correctly.

Next, we flesh out a (perhaps primitive) input module and an output module. Voilà! A running system that does something, however dull. Now, function by function, we incrementally build and add modules. At every stage we have a running system. If we are diligent, we have at every stage a debugged, tested system. (As the system grows, so does the burden of regression-testing each new module against all the previous test cases.)

After every function works at a primitive level, we refine or rewrite first one module and then another, incrementally growing the system. Sometimes, to be sure, we have to change the original driving loop, and or even its module interfaces. (p. 267-8)

Customer collaboration over contract negotiation

The following quote illustrates the third principle of the Agile Manifesto by referring to the important role played by customers and users in software development processes:

Since we have a working system at all times

we can begin user testing very early, and

we can adopt a build-to-budget strategy that protects absolutely against schedule or budget overruns (at the cost of possible functional shortfall). (p. 268)

Responding to change over following a plan

This principle of the Agile Manifesto is reflected by Brooks, once again, by referring to one of the problems with the waterfall model, highlighting the importance of legitimizing learning processes in software development processes.

The basic fallacy of the waterfall model is that it assumes a project goes through the process once, that the architecture is excellent and easy to use, the implementation design is sound, and the realization is fixable as testing proceeds. Another way of saying it is that the waterfall model assumes the mistakes will all be in the realization, and thus that their repair can be smoothly interspersed with component and system testing.

...

Designing the implementation will show that some architectural features cripple performance; so the architecture has to be reworked. Coding the realization will show some functions to balloon space requirements; so there may have to be changes to architecture and implementation.

One may well, therefore, iterate through two or more architecture–implementation design cycles before realizing anything as code. (p. 266–7)

4. Conclusion

The analysis of software development processes presented in this essay suggests that agile software development can be viewed as a convergence point of many ideas with which the SE community has been familiar for many years. In other words, Brooks' ideas, which were published in three consecutive decades, can be viewed as intermediate stages of a process of convergence towards the establishment of the paradigm of agile software development, whose extensive application began around the year 2000.

An interesting question to be asked at this stage is: Why did it take so long to establish the agile approach, whose seeds could be found in Brooks' manuscript as early as 1975, over thirty years ago. My speculation is rooted in the community's needs for a concept that encapsulates a different software development approach. Introduction of the term agile software development and formulation of the Agile Manifesto in 2001 provided such an anchor as well as a unifying concept for the community of software developers in general, and particularly for

the different voices that called for change in software development processes at the end of the 20th century and beginning of the 21st century (see Hazzan, Seger and Luria, submitted).

5. References

- Beck, K. (2000). Extreme Programming Explained: Embrace Change, Addison-Wesley.
- Beck, K. (2002). Test Driven Development: By Example, Addison-Wesley Professional.
- Beck, K. and Fowler, M. (2000). Planning Extreme Programming, Addison-Wesley.
- Brooks, J. F. P. (1975). The Mythical Man-Month, Addison-Wesley, Reading, MA.
- Brooks, J. F. P. (1987). No silver bullet: Essence and accidents of software engineering, Computer 20, pp. 10 - 19.
- Brooks, J. F. P. (1995). The Mythical Man-Month (anniversary ed.), Addison-Wesley Longman Publishing Co., Inc.
- Fowler, M. (1999). Refactoring - Improving the Design of Existing Code, Addison-Wesley.
- Hazzan, O., Seger, T. and Luria, G. (submitted). How did the creators of the Agile Manifesto turn from technology leaders to leaders of a cultural change?

A Compacting Real-Time Memory Management System

Silviu S. Craciunas Christoph M. Kirsch Hannes Payer
Ana Sokolova Horst Stadler Robert Staudinger

Department of Computer Sciences
University of Salzburg, Austria

firstname.lastname@cs.uni-salzburg.at

Abstract

We propose a real real-time memory management system called Compact-fit that offers both time and space predictability. Compact-fit is a compacting memory management system for allocating, deallocating, and accessing memory in real time. The system provides predictable memory fragmentation and response times that are constant or linear in the size of the request, independently of the global memory state. We present two Compact-fit implementations and compare them to established memory management systems, which all fail to provide predictable memory fragmentation. The experiments confirm our theoretical complexity bounds and demonstrate competitive performance. In addition, we can control the performance versus fragmentation trade-off via our concept of partial compaction. The system can be parameterized with the needed level of compaction, improving the performance while keeping memory fragmentation predictable.

1 Introduction

We present a compacting real-time memory management system called Compact-fit (CF) together with a moving

and a non-moving implementation. Compact-fit is an explicit memory management system for allocating, deallocating, and accessing memory objects. Memory fragmentation in CF is bounded by a compile-time parameter. In CF compaction may only happen upon freeing a memory object and involves moving a single memory object of the same size.

Memory in CF is partitioned into 16KB pages. Each page is an instance of a so-called size-class, which partitions a page further into same-sized page-blocks. We adapt the concept of pages and size-classes from [2]. A memory object is always allocated in a page of the smallest-size size-class whose page-blocks still fit the object. Memory objects larger than 16KB are currently not supported. However, in a future version, arraylets [3] may be used to handle objects of larger size with CF's complexity bounds.

The key idea in CF is to keep the memory size-class-compact at all times. In other words, at most one page of each size-class may be not-full at any time while all other pages of the size-class are always kept full. Whenever a memory object is freed, a memory object in the not-full page is moved to take its place and thus maintain the invariant. If the not-full page becomes empty, it can be reused in any size-class. Using several list and bitmap data structures, free space can be found in constant time, upon an allocation request.

The moving CF implementation maps page-blocks directly to physically contiguous pieces of memory, and therefore requires moving memory objects for compaction. Allocation takes constant time in the moving implementation, whereas deallocation takes linear time if compaction occurs. Dereferencing requires an additional pointer indirection and takes constant time.

The non-moving CF implementation uses a block table (effectively creating a virtual memory) to map page-blocks into physical block-frames that can be located anywhere in memory. In this case, compaction merely requires re-programming the block table rather than moving memory objects. However, although compaction may be faster, deallocation still takes linear time in the size of the object due to the block table administration. For the same reason allocation also takes linear time in the non-moving implementation. Our experiments show that deallocation is faster in the non-moving implementation for configurations in which block-frames are at least 80B.

Dereferencing requires two additional pointer indirection and takes constant time.

A pointer in CF is an address and an offset. The CF system therefore supports offset-based rather than address-based pointer arithmetics, which we elaborate on later in the paper. Note that, in principle, the moving implementation may also support address-based pointer arithmetics since each memory object is allocated in a single physically contiguous piece, that may however move during compaction.

In both implementations we can relax the always-compact-requirement allowing for more than one not-full page per size-class. As a result deallocation takes less time: it reduces up to constant time. This way we formalize, control, and implement the trade-off between temporal performance and memory fragmentation.

We present the results of benchmarking both implementations, as well as implementations of non-compacting real-time memory management systems (Half-fit [11] and TLSF [10]) and traditional (non-real-time) memory management systems (First-fit [7], Best-fit [7], and Doug Lea's allocator [8]) using synthetic workloads.

The contributions of this paper are: the CF system, the moving and non-moving implementations, and the experimental results on bare metal and Linux.

The rest of the paper is organized as follows: Section 2 discusses the principles behind the design of the compacting real-time memory management system. The implementation details and the complexity issues are presented in Section 3. We discuss related work in Section 4, and present the experiments, results and comparisons in Section 5. Section 6 wraps up with discussion and conclusion.

2 Principles of Compact-Fit

We start by introducing the goals of memory management in general and the requirements for real-time performance in particular. Having the basis set, we present our proposal for a compacting memory management system that meets the real-time requirements. We focus on the conceptual issues in this section and on the technical details in the following section.

2.1 Basics of Memory Management

By memory management we mean dynamic memory management. Applications use the dynamic memory management system to allocate and free (deallocate) memory blocks of arbitrary size in arbitrary order. In addition, applications can use the memory management system for accessing already allocated memory, dereferencing. Memory deallocation can lead to memory holes, which can not be reused for future allocation requests, if they are too small: this is the fragmentation problem. The complexity of allocation and deallocation depends on the fragmentation. A way to fight fragmentation is by performing compaction or defragmentation: a process of rearranging the used memory space so that larger contiguous pieces of memory become available.

There are two types of dynamic memory management systems:

- * explicit, in which an application has to explicitly invoke the corresponding procedures of the dynamic memory management system for allocating and deallocating memory, and
- * implicit, in which memory deallocation is no longer explicit, i.e., allocated memory that is not used anymore is detected and freed automatically. Such systems are called garbage collectors.

In this paper we propose an explicit dynamic memory management system.

2.2 Real-Time Requirements

Traditional dynamic memory management strategies are typically non-deterministic and have thus been avoided in the real-time domain. The memory used by real-time programs is usually allocated statically, which used to be a sufficient solution in many real-time applications. Nowadays increasing complexity demands greater flexibility of memory allocation, so there is a need of designing dynamic real-time memory management systems. Even in soft real-time systems and general purpose operating systems there exist time-critical components such as device drivers that operate on limited amount of memory because of resource and/or security constraints and may require predictable memory management.

In an ideal dynamic real-time memory management system each unit operation (memory allocation, deallocation, and dereferencing) takes constant time. We refer to this time as the response time of the operation of the memory management. If constant response times can not be achieved, then bounded response times are also acceptable. However, the response times have to be predictable, i.e., bounded by the size of the actual request and not by the global state of the memory.

More precisely, real-time systems should exhibit predictability of response times and of available resources. The fragmentation problem affects the predictability of the response times. For example, if moving objects in order to create enough contiguous space is done upon an allocation request, then the response time of allocation may depend on the global state of the memory.

Predictability of available memory means that the number of the actual allocations together with their sizes determines how many more allocations of a given size will succeed before running out of memory, independently of the allocation and deallocation history. In a predictable system also the amount of fragmentation is predictable and depends only on the actual allocated objects. In addition to predictability, fragmentation should be minimized for better utilization of the available memory.

Most of the established explicit dynamic memory management systems [8,7] are optimized to offer excellent best-case and average-case response times, but in the worst-case they are unbounded, i.e., they depend on the global state of the memory. Hence these systems do not meet the above mentioned requirement on predictability of response times.

Moreover, to the best of our knowledge, none of the established explicit dynamic memory management systems meets the memory predictability requirement since fragmentation depends on the allocation and deallocation history.

The memory management system that we propose offers bounded response times (constant or linear in the size of the request) and predictable memory fragmentation, which is achieved via compaction.

Performing compaction operations could be done in either event- or time-triggered manner. As the names suggest, event-triggered compaction is initiated upon the occurrence of a significant event, whereas time-triggered compaction happens at predetermined points in time. Our compaction algorithm is event-triggered, compaction may be invoked upon deallocation.

2.3 Abstract and Concrete Address Space

We now describe the compacting real-time memory management system. At first, we focus on the management of memory addresses. Conceptually, there are two memory layers: the abstract address space and the concrete address space. Allocated objects are placed in contiguous portions of the concrete address space. For each allocated object, there is exactly one abstract address in the abstract address space. No direct references from applications to the concrete address space are possible: an application references the abstract address of an object, which furthermore uniquely determines the object in the concrete space. Therefore the applications and the memory objects (in the concrete space) are decoupled. All memory operations operate on abstract addresses. We start by defining the needed notions and notations.

The abstract address space is a finite set of integers denoted by \mathbb{A} . An abstract address a is an element of the abstract address space, $a \in \mathbb{A}$.

The concrete address space is a finite interval of integers denoted by \mathbb{C} . Note that, since it is an interval, \mathbb{C} is contiguous. Moreover, both the concrete and the abstract address spaces are linearly ordered by the standard ordering of the integers. A concrete address c is an element of the concrete address space, $c \in \mathbb{C}$.

A memory object m is a subinterval of the concrete address space, $m \in \mathcal{I}(\mathbb{C})$. For each memory object, two concrete addresses $c_1, c_2 \in \mathbb{C}$, such that $c_1 \leq c_2$, define its range, i.e., we have $m = [c_1, c_2] = \{x \mid c_1 \leq x \leq c_2\}$.

As mentioned above, a used abstract address refers to a unique range of concrete addresses, which represents a

$\{\mathrm{pointer}\}(1,1) = 3\$$ and $\{\mathrm{pointer}\}(3,1) = 9\$$.

We now elaborate the benefits of using an abstract memory space. All references are redirected via the abstract memory space. An application points to a concrete memory location via an abstract pointer, cf.

Large data-structures often consist of a number of allocated memory objects connected via references (e.g. linked lists or trees) that depict the dependencies between the objects. These dependencies are handled via abstract pointers as well.

Indirect referencing facilitates predictability of reference updates during compaction. If fragmentation occurs, the concrete address space \mathbb{C} gets compacted and the references from the abstract address space \mathbb{A} to the concrete address space \mathbb{C} are updated. Hence, objects are moved in \mathbb{C} and references are updated in \mathbb{A} . The number of reference updates is bounded: movement of one memory object in \mathbb{C} leads to exactly one reference update in \mathbb{A} . In contrast, direct referencing (related to object dependencies) would imply unpredictable number of reference updates. This is why we chose for an abstract address space design. However, note that the abstract address space is only required for predictable reference updating during compaction but otherwise completely orthogonal to the compaction algorithm and its moving and non-moving implementations described below.

Image proxyFragmentation3

(a) Before: fragmented concrete space

Image proxyCompaction3

(b) After: compact concrete space

The CF system provides three explicit memory operations (whose implementation we discuss in Section 3):

* $\text{malloc}(\text{size})$ is used to create a memory object of a given size. It takes an integer $\text{size} > 0$ as argument and returns an abstract pointer $a_{\{p\}} = (a, o)$, where a is an abstract address that references to the allocated memory object and the offset o is set to 0, the beginning of the memory object.

* $\text{free}(a)$ takes an abstract address a as argument and frees the memory object that belongs to this abstract address. The abstract address mapping is released.

* $\text{dereference}(a_{\{p\}})$ returns the concrete address c of an abstract pointer $a_{\{p\}} = (a, o)$, where a is the abstract address of a memory object and the offset o points to the actual position within the memory object.

Note that the abstract address of an allocated memory object never changes until the object gets freed. The abstract address can therefore be used for sharing objects. The concrete address(es) of an allocated memory object may change due to compaction. To this end, we point out another difference between the abstract and the concrete address space. Over time, they may both get fragmented. The fragmentation of the concrete space presents a problem since upon an allocation request the memory management system must provide a sufficiently large contiguous memory range. In the case of the abstract address space, a single address is used per memory object, independently of its size. Hence, upon an allocation request, the memory management system needs to find only a single unused abstract address. We can achieve this within a constant-time bound, without keeping the abstract address space compact.

2.4 Size-Classes

For administration of the concrete address space, we adopt the approach set for Metronome [2,3,1].

The following ingredients describe the organization of the concrete address space.

* Pages: The memory is divided into units of a fixed size P , called pages. For example, in our implementation each page has a size $P = 16\text{KB}$.

* Page-blocks: Each used page is subdivided into page-blocks. All page-blocks in one page have the same size. In total, there are n predefined page-block sizes S_1, \dots, S_n where $S_i < S_j$ for $i < j$.

Hence the maximal page-block size is S_n .

* Size-classes: Pages are grouped into size-classes. There are n size-classes (just as there are n page-block sizes). Let $1 \leq i \leq n$. Each page with page-blocks of size S_i belongs to the i -th size-class. Furthermore, each size-class is organized as a doubly-circularly-linked list.

Every allocation request is handled by a single page-block. When an allocation request $\text{malloc}(\text{size})$ arrives, CF determines the best fitting page-block size S_i and inserts the object into a page-block in a page that belongs to size-class i . The best fitting page-block size is the unique page-block size S_i that satisfies $S_{i-1} < \text{size} \leq S_i$.

If a used page becomes free upon deallocation, then the page is removed from its size-class and can be reused in any possible size-class.

2.5 Fragmentation

The size-classes approach is exposed to several types of fragmentation: page-block-internal fragmentation, page-internal fragmentation, and size-external fragmentation [2]. We will briefly discuss each of them and the impact they have on our design decisions. Figure 3 shows the different types of fragmentation as well.

2.5.1 Page-Block-Internal Fragmentation

Page-block-internal fragmentation is the unused space at the end of a page-block. Given a page p in size class i (the page-blocks in p have size S_i), let b_j for $j = 1, \dots, B_p$ be the page-blocks appearing in the page p , where $B_p = \lfloor \text{page_size} / S_i \rfloor$. For a page-block b_j we define $\text{used}(b_j) = 1$ if b_j is in use, and $\text{used}(b_j) = 0$ otherwise. We also write $\text{data}(b_j)$ for the amount of memory of b_j that is allocated. The page-block-internal fragmentation for the page p is calculated as

$$F_B(p) = \sum_{j=1}^{B_p} \text{used}(b_j) \cdot (S_i - \text{data}(b_j)).$$

One can also calculate the total page-block-internal fragmentation in the memory by summing up the page-block-internal fragmentation for each page.

The total page-block-internal fragmentation can be bounded by a factor f if the page-block sizes are chosen carefully. Namely, Berger et al. [4] suggest the following ratio between adjacent page-block sizes:

$$S_k = \lceil S_{k-1}(1+f) \rceil \quad (1)$$

for $k = 2, \dots, n$. The size of the smallest page-blocks S_1 and the parameter f can be chosen program-specifically. Bacon et al. [2] propose a value for the parameter $f = 1/8$, which leads to minor size differences for smaller size-classes and major size differences for larger size-classes.

2.5.2 Page-Internal Fragmentation

Page-internal fragmentation is the unused space at the end of a page. If all possible page-block sizes S_1, \dots, S_n are divisors of the page size P , then there is no page-internal fragmentation in the system. However, if one uses Equation (1) for the choice of page-block sizes, then one also has to acknowledge the page-internal fragmentation. For a page p in size-class i , it is defined as

$$F_P(p) = P \bmod S_i$$

The total page-internal fragmentation is the sum of $F_P(p)$ taken over all used pages p .

2.5.3 Size-External Fragmentation

Size-external fragmentation measures the unused space in a used page. This space is considered fragmented or "wasted" because it can only be used for allocation requests in the given size-class. For example, let p be a page in size-class i with $S_i = 32B$. If only one page-block of p is allocated, then there is $P - 32B$ unused memory in this page. If no more allocation requests arrive for this size-class, then this unused memory can never be used again. In such a situation an object of size 32B consumes the whole page. The size-external fragmentation of a page p in size-class i is bounded by

$$F_S(p) = P - S_i$$

The total size-external fragmentation in the system is bounded by the sum of $F_S(p)$, over all pages.

The more size-classes there are in the system, the less page-block-internal fragmentation occurs, but therefore the size-external fragmentation may grow. Hence, there is a trade-off between page-block-internal and size-external fragmentation, which must be considered when defining the size-classes.

2.6 The Compaction Algorithm

The compaction algorithm of CF behaves as follows. Compaction is performed after deallocation in the size-class affected by the deallocation request. It implies movement of only one memory object in the affected size-class. Before presenting the algorithm, we state two invariants and two rules that are related to our compaction strategy. Recall that each size-class is a doubly-circularly-linked list.

Invariant 1 In each size-class there exists at most one page which is not full.

Invariant 2 In each size-class, if there is a not-full page, then this is the last page in the size-class list.

The compaction algorithm acts according to the following two rules.

Rule 1 If a memory object of a full page p in a size-class gets freed, and there exists no not-full page, then p becomes the not-full page of its size-class and it is placed at the end of the size-class list.

Rule 2 If a memory object of a full page p in a size-class gets freed, and there exists a not-full page p_n in the size-class, then one memory object of p_n moves to p . If p_n becomes empty, then it is removed from the size-class.

Not every deallocation request requires moving of a memory object. The cases when no moving is necessary are:

* The deallocated memory object is in the unique not-full page of the size-class. This case imposes no work except when the deallocated memory object is the only memory object in the page. Then the page is removed from the size-class.

* There is no not-full page in the size-class where deallocation happened. In this case only a fixed number of list-reference updates is needed in order that the affected page becomes the last page in the size-class list.

Note that when a memory object moves from one page to another, then we need to perform a reference update in the abstract address space, in order that the abstract address of a memory object points to its new location.

The compaction algorithm is presented in Listing 1.

```
\begin{figure}\renewcommand{baselinestretch}{1} \begin{lstlisting}[frame=tb,ca... ..e);
abstract_address_space_update(object); } } } \end{lstlisting}\end{figure}
```

3 Details and Complexity

In this section we elaborate on the implementation details of CF. We start by discussing the details of the concrete address space management, i.e., administration of the page-blocks, pages, and size-classes. Next we describe the implementation and the complexity results for both the moving and the non-moving version of CF. At the end of this section we also explain the concept of partial compaction.

3.1 Managing Size-Classes

As mentioned above, we use pages of size 16KB. If needed, the page size can be modified for specific applications. The minimal page-block size S_1 in our benchmarks is 32B, but it can also be reduced to a smaller size. Successive page-block sizes are determined by Equation (1) taking $f = 1/8$.

In addition to the 16KB storage space, each page has a header with additional information used for the memory

management. The layout of a page with all administrative information is illustrated in Figure 4.

Figure 4: Page layout

Image page4

The fields Next Page and Previous Page contain references to the next and the previous page within the size-class, respectively. These two references build the size-class list. The field Size-Class refers to the size-class instance of the page, which further refers to the head of the size-class list. Hence, we can directly access the head and therefore also the tail of the size-class list from any page, which is important for compaction.

A page consists of page-blocks, some of which are in use and some are free. For memory object allocation, we must find a free page-block in a page in constant time and for compaction, we must find a used page-block in a page in constant time. Therefore, the state of the page (the status of all page-blocks) is recorded in a two-dimensional bitmap, as shown in Figure 4. A bitmap of size 16×32 is enough to record the status of the page-blocks since we have at most 512 page-blocks per page. In addition, we need 16 more bits to record if at least one bit is set for each row of the bitmap. This additional bitstring is essential for constant-time access to a used page-block within the page and it is used to determine whether a page is full or not in constant time (one comparison). There are CPU instructions that find a set bit in a bitstring in constant time. These instructions are limited to bitstrings of length 32 (on a 32-bit CPU), which is the reason why we use such a two-dimensional bitmap. In order to get a used (respectively free) page-block we first find a set bit in the additional bitstring, and then get a set bit in the corresponding row of the bitmap.

The free pages are organized in a special LIFO list. Since all pages are arranged contiguously in memory, no list initialization is necessary. If the LIFO list is empty and a free element is required, the next unused element of the contiguous space is returned (if such an element exists). We refer to this list construction as free-list.

Note that the administrative information of a page, as shown in Figure 4, takes 78B out of the 16KB page memory. Hence, the memory overhead is less than 0.47%.

3.2 Moving Implementation

In this version, memory objects are moved in physical memory during compaction.

The abstract address space is implemented in a contiguous piece of memory. The free entries of the abstract address space are organized in a free-list.

The concrete address space is organized as described in Section 3.1. Each page is implemented as a contiguous piece of memory as well. Moreover, each page-block contains an explicit reference to its abstract address in the abstract address space. This reference is located at the end of the page-block. In the worst case, it occupies 12.5% of the page-block size. These backward references allow us to find in constant time the abstract address of a memory object of which we only know its concrete address. Therefore they are essential for constant-time update of the abstract address space during compaction.

We next present the allocation, deallocation, and dereferencing algorithms and discuss their complexity. The algorithm for allocation `cfm_malloc` is presented in Listing 2. The function `get_page_of_size_class` returns a page of the corresponding size-class in constant time: if all pages in the size-class are full, then with help of the free-list of free pages, we get a new page; otherwise the not-full page of the size-class is returned. Hence this function needs constant time. The function `get_free_page_block` takes constant time, using the inverse of the two-dimensional bitmap of a page. Declaring a page-block used is just a bit-set operation. As mentioned above, the free abstract addresses are organized in a free-list, so the function `create_abstract_address` takes constant time. As a result, `cfm_malloc` takes constant time, i.e., $\Theta(\text{cfm_malloc}(\text{size})) = \Theta(1)$.

```
\begin{figure}\renewcommand{baselinestretch}{1} \begin{lstlisting}[frame=tb,ca... ...ge]; return  
create_abstract_address(page_block); \end{lstlisting}\end{figure}
```

The deallocation algorithm `cfm_free` is shown in Listing 3. The function `get_page_block` takes constant time, since it only accesses the memory location to which the abstract address refers. The function `get_page` takes several arithmetic operations, i.e., constant time. Namely, pages are linearly aligned in memory so for a given page-block

we can calculate the starting address of its page. The function `get_size_class` is executed in constant time, since every page contains a field `Size-Class`. The function `set_free_page_block` changes the value of a single bit in the bitmap, so it also requires constant time and `add_free_abstract_address` amounts to adding a new element to the corresponding free-list, which is done in constant time too. Removing a page from a size-class `remove_page` requires also constant time: first a page is removed from the size-class list; then it is added to the free-list of empty pages. Therefore, the complexity of `cfm_free` equals the complexity of the compaction algorithm.

The complexity of the compaction algorithm, `compaction`, is linear in the size of the page-blocks in the corresponding size-class since it involves moving a memory object. Note that the complexity of the abstract address space update is constant, due to the direct reference from page-blocks to abstract addresses.

Hence, the worst-case complexity of `cfm_free` is linear in the size of the page-block: $O(\text{cfm_free}(\text{abs_address}) \cdot s) = O(s)$ for s being the size of page-blocks in the size-class where `abs_address` refers to. Thus, for a fixed size-class, we have constant complexity.

```
\begin{figure}\renewcommand{baselinestretch}{1} \begin{lstlisting}[frame=tb,ca... ..ge); } else  
{ compaction(size_class, page); } } \end{lstlisting}\end{figure}
```

In this moving implementation, the physical location of a memory object is accessed by dereferencing an abstract pointer. The dereferencing contains a single line of code `*(abs_address + offset)`; given an abstract pointer `(abs_address, offset)`.

To conclude, the only source of non-constant (linear) complexity in the moving implementation is the moving of objects during compaction. In an attempt to lower this bound by a constant factor, we implement the non-moving version.

3.3 Non-Moving Implementation

We call this implementation non-moving, since memory objects do not change their location in physical memory

throughout their lifetime, even if compaction is performed.

In the non-moving implementation, the abstract address space is still a contiguous piece of memory. However, we no longer use the free-list for administrating free abstract addresses, since now there is an implicit mapping from the memory objects to the abstract addresses. We will elaborate on this in the next paragraph. The implicit references (from memory objects to abstract addresses) are used for constant-time updates of the abstract address space.

First, let us explain the difference in the implementation of the concrete address space. The concrete address space is managed by a virtual memory. The virtual memory consists of blocks of equal size. The physical memory is contiguous and correspondingly divided into block-frames of equal size. For further reference we denote this size by s_b . These blocks and block-frames must not be confused with the page-blocks: they are global, for the whole memory, above the page concept. Therefore, each block-frame is directly accessible by its ordinal number. The free block-frames are also organized in a free-list. The abstract address space is pre-allocated, and contains as many potential abstract addresses as there are block-frames. A unique abstract address corresponds to a memory object m : the abstract address at position k in the abstract address space, where k is the ordinal number of the first block-frame of m in the physical memory. This way, we do not need an explicit reference stored in each page-block, thus we avoid the space overhead characteristic to the moving implementation. Moreover, getting a free abstract address is immediate, as long as we can allocate a block-frame in physical memory.

A block table records the mapping from virtual memory blocks to memory block-frames. In our implementation, the block table is distributed over the pages, which simplifies the page-specific operations. The organization of the memory in this implementation is shown in Figure 5.

Figure 5: Memory layout of the non-moving implementation

Image proxyImplicit1

Objects are allocated in contiguous virtual memory ranges, but actually stored in arbitrarily distributed physical memory block-frames. We still use the concepts of pages, size-classes and page-blocks, with some differences. A page is now a virtual page: It does not contain any data; in its Data segment it contains the part of the block table that points to the actual data in the physical memory. Moreover, for administration purposes all page-block sizes are multiples of the unique block size. Hence, each page-block consists of an integer number of blocks. This implies that we can not directly use Equation (1), we need to round-up the page-block sizes to the next multiple of the block size.

The allocation algorithm is shown in Listing 4. In comparison to the moving implementation, we have now a loop that handles the allocation block-frame-wise. Note that `number_of_blocks(page_block)` is constant for a given size-class. Getting a free block-frame, and creating a corresponding block-table entry, takes constant time. Therefore, the complexity of the allocation algorithm is linear in the number of block-frames in a page-block, i.e., $\Theta(\text{scfm_malloc}(\text{size})) = \Theta(n)$ where $n = s/s_b$ for s the page-block size of the size-class. Again, this means constant complexity in the size-class.

The function `create_abstract_address` in the non-moving implementation uses the implicit references from memory objects to abstract addresses.

Listing 5 shows the deallocation algorithm.

```

\begin{figure}\renewcommand{baselinestretch}{1} \begin{lstlisting}[frame=tb,ca...    ...;    }    return
create_abstract_address(page_block); } \end{lstlisting}\end{figure}

\begin{figure}\renewcommand{baselinestretch}{1} \begin{lstlisting}[frame=tb,ca...    ...ge);    }    else
{ compaction(size_class, page); } } \end{lstlisting}\end{figure}

```

In comparison to the moving implementation, we have to free each block-frame in the memory occupied by the freed memory object. This requires a loop with as many iterations as there are block-frames for the memory

object. As above, there are $n = s/s_b$ block-frames, where s is the size of the size-class and s_b the size of the block-frame. Moreover, the compaction algorithm is implemented differently: memory objects are only virtually moved, by updating the block table. This is still linear in the size of the size-class, but we achieve a constant speed-up: it actually takes n updates of the block table. As a result, the complexity of the deallocation algorithm in the non-moving implementation is $\Theta(s_{\text{free}}) = \Theta(n)$, which is again constant for a given size-class.

Finally, we consider the dereference algorithm. It provides direct access to a memory location corresponding to a given abstract pointer (abs_address, offset). Dereferencing takes constant time, with several more calculations than in the moving implementation. The algorithm is shown in Listing 6. In the code, s_b stands for the size of a block, s_b .

```
\begin{figure}\renewcommand{baselinestretch}{1} \begin{lstlisting}[frame=tb,ca... ...address + (offset div s_b)) + (offset mod s_b); } \end{lstlisting}\end{figure}
```

We conclude that the non-moving implementation achieves a constant speed-up of the compaction algorithm, while the complexity of the allocation algorithm grows from constant to linear. The complexity of deallocation and dereference is the same in both implementations.

3.4 Partial Compaction

Up to now we always considered the very strong aim of "always compact size-classes", i.e., our invariant was that at any moment in time in each size-class at most one page is not full. We now relax this invariant by allowing for a given number of not-full pages per size-class. Varying the number of allowed not-full pages per size-class, max_nr_nf_pages , results in various levels of compaction. For example, $\text{max_nr_nf_pages} = 1$ means always compact memory (as described above), but therefore larger compaction overhead, whereas high values of max_nr_nf_pages lead to faster memory management, for the price of higher fragmentation. This way we formalize, control, and implement the trade-off between temporal performance and memory fragmentation.

We implement this new approach as follows. A size-class instance now consists of three fields: `head_full_pages`, `head_not_full_page`, and `nr_not_full_pages`. Therefore it consists of two doubly-circularly-linked lists: one containing the full pages and another one containing the not-full pages. The initial value of `nr_not_full_pages` is zero and it never exceeds a pre-given number, `max_nr_nf_pages`. If all pages are full, then allocation produces a new not-full page, so `nr_not_full_pages` is incremented. In case a not-full page gets full after allocation, it is moved to the list of full pages, and `nr_not_full_pages` is decremented. If deallocation happens on a page-block that is in a not-full page, no compaction is done. If it happens on a page-block which is in a full page, then compaction is called if `nr_not_full_pages` \leq `max_nr_nf_pages`. Otherwise, no compaction is done, the affected page is moved from the list of full pages to the list of not-full pages, and `nr_not_full_pages` is incremented.

For better average-case temporal and spatial performance, we keep pages that are more than half-full at the end of the not-full list, and pages that are at most half-full at the head of the list. Allocation is served by the last page of the not-full list, which may increase the chances that this page gets full. Used page-blocks that need to be moved because of compaction are taken from the first page of the not-full list, which might increase the chances that this page gets empty. Note that, for the best possible spatial performance, it would be better to keep the not-full list sorted according to the number of free page-blocks in each non-full page. However, inserting a page in a sorted list is too time-consuming for our purposes and can not be done in constant time.

It should be clear that each deallocation when `nr_not_full_pages` \leq `max_nr_nf_pages` takes constant time, i.e., involves no compaction. However, this guarantee is not very useful in practice: given a mutator we can not find out (at compile time) the maximal number of not-full pages it produces. Therefore we describe another guarantee.

Given a size-class, let `n_f` count deallocations for which no subsequent allocation was done. Initially, `n_f=0`. Whenever deallocation happens, `n_f` is incremented. Whenever allocation happens, `n_f` is decremented, unless it is already zero. We can now state the following guarantee.

Proposition 1 Each deallocation that happens when `n_f < max_nr_nf_pages - 1` takes constant time in the CF moving implementation, i.e., it involves no compaction.

Namely, a simple analysis shows that $\text{nr_not_full_pages} \leq n_f + 1$ is an invariant for a given size-class. It holds initially since then $\text{nr_not_full_pages} = n_f = 0$. Each allocation and deallocation keeps the property valid. Therefore, if a deallocation happens when $n_f < \text{max_nr_nf_pages} - 1$, then $\text{nr_not_full_pages} < \text{max_nr_nf_pages}$ and hence compaction is not called.

Program analysis can be used to determine the maximum value of n_f for a given mutator at compile time. More advanced program analysis (e.g. analysis of sequences of allocations and deallocations) might provide us with a stronger guarantee than the one above. Employing program analysis is one of our future-work aims. The effect of partial compaction can be seen in the experiments in Section 5.

3.5 Pointer Arithmetic

Since we make the distinction between abstract and concrete address space, our malloc function returns a reference to an abstract address, instead of a reference to a memory location. Therefore, pointer arithmetic needs adjustment, so that it fits our model. In order to enable standard pointer arithmetic, we need the structure of an abstract pointer. It contains a field `abs_address` and a field `offset`. Consider the example of C code in Listing 7. The same is achieved in CF by the code presented in Listing 8.

```
\begin{figure}\renewcommand{baselinestretch}{1} \begin{lstlisting}[frame=tb,ca... ...*value; value = malloc(40);  
value++; print(*value); \end{lstlisting}\end{figure}
```

```
\begin{figure}\renewcommand{baselinestretch}{1} \begin{lstlisting}[frame=tb,ca... ...int(dereference(value.abs_address,  
value.offset)); \end{lstlisting}\end{figure}
```

A virtual machine (e.g. a Java VM) can encapsulate the abstract pointer concept. C programs which use CF instead of a conventional memory management system have to be translated into code that uses the abstract pointer concept. An automatic translation tool is left for future work.

On an Intel architecture running Linux, GCC translates a standard pointer dereference (including offset addition) to 6 assembler instructions, whereas a CFM abstract pointer dereference results in 7 assembler instructions (one additional pointer dereference instruction is needed). A CFNM abstract pointer dereference results in 11 assembler instructions. The additional 5 assembler instructions consist of one dereference operation and 4 instructions that calculate the memory target address and move the pointer to that address.

4 Related Work

In this section we briefly discuss existing memory management systems (mostly for real time). We distinguish explicit and implicit memory management systems and consider both non-compacting real-time, and non-real-time memory management systems. Jones [6] maintains an extensive online bibliography of memory management publications.

4.1 Explicit Memory Management

There are several established explicit memory management systems: First-fit [7], Best-fit [7], Doug Lea's allocator (DL) [8], Half-fit [11], and Two-level-segregated-fit (TLSF) [10]. A detailed overview of the explicit memory management systems can be found in [9,13].

First-fit and Best-fit are sequential fit allocators, not suitable for real-time applications. In the worst case, they scan almost the whole memory in order to satisfy an allocation request. DL is a non-real-time allocator used in many systems, e.g. in some versions of Linux.

Half-fit and TLSF offer constant response-time bounds for allocation and deallocation. However, both approaches may suffer from unbounded memory fragmentation.

None of the above mentioned algorithms perform compaction. Instead, they attempt to fight fragmentation by clever allocation, and therefore can not give explicit fragmentation guarantees.

In Section 5 we present a comparison of the CF implementations with First-fit, Best-fit, DL, Half-fit, and TLSF.

4.2 Implicit Memory Management

We elaborate on two established implicit real-time memory management systems: Jamaica [14] and Metronome [2].

Jamaica splits memory objects into fixed-size blocks that can be arbitrarily located in memory and connected in a linked list (or tree) structure. Allocation and deallocation achieve the same bounds like our non-moving implementation. Dereferencing in Jamaica involves going through the list of memory object blocks, therefore it takes linear (or logarithmic) time in the size of the object. Compaction is not needed for Jamaica, since memory objects do not occupy contiguous pieces of memory.

Metronome is a time-triggered garbage collector. As mentioned above, we adapt some concepts like pages and size-classes from Metronome. Compaction in Metronome is part of the garbage collection cycles. The time used for compaction is estimated to at most 6% of the collection time [2].

5 Experiments and Results

In this section, we benchmark the moving (CFM) and non-moving (CFNM) implementations as well as the partial compaction strategy of CF in a number of experiments. Moreover, we compare both CF implementations with the dynamic memory management algorithms First-fit, Best-fit, DL, Half-fit, and TLSF. The implementations of First-fit, Best-fit, Half-fit, and TLSF we borrow from Masmano et al. [9]. We took the original implementation of DL from Doug Lea's web page [8].

5.1 Testing Environment

We have performed processor-instruction measurements of our algorithms on a standard Linux system, and bare-metal execution-time measurements on a Gumstix connex400 board [5] running a minimal hardware abstraction layer (HAL).

Processor-instruction measurements eliminate interferences like cache effects. For measurement purposes, our

mutators are instrumented using the ptrace [12] system call. The processor-instruction and execution-time measurements are almost the same except that the former are cleaner, free of side effects. For this reason, we present the processor-instruction results only.

Our mutators provide synthetic workloads designed to create worst-case and average-case scenarios. We have not obtained standardized macrobenchmark results for lack of an automatic code translator or virtual machine implementation that incorporate the abstract pointer concept.

5.2 Results: Incremental Tests

In this benchmark, we run a mutator with incremental behavior: it allocates memory objects of increasing size starting from 8B increasing by 4B until the memory gets full at 7MB. Then, it deallocates each second object. We measure this process in the deallocation experiments. Finally, the mutator allocates the deallocated objects once more. We measure this process in the allocation experiments. In Figure 6, the \$ x\$-axes show the number of invoked memory operations, whereas the \$ y\$-axes represent the corresponding measured number of executed instructions.

Figure 6: Incremental microbenchmark

Image all_malloc_incremental_instr

(a) Allocation

Image all_free_incremental_instr

(b) Deallocation with compaction

Image all_free_incremental_no_compaction_instr

(c) Deallocation, partial compaction

Figure 6(a) shows the number of processor instructions for allocation. The behavior of First-fit and Best-fit is highly unpredictable. DL appears more bounded. Half-fit and TLSF perform allocation operations fast and in

constant time. The behavior of the CF implementations is according to our theoretical results: constant for CFM and linear for CFNM. Note that the y -axes of the graphs have logarithmic scale. Both CF implementations are bounded but slower than Half-fit and TLSF due to the additional administrative work for (potential) compaction. CFM is as fast as DL, and faster than First-fit and Best-fit. The average number of instructions for allocation with CFM is 169.61, the standard deviation is 8.63.

The deallocation benchmark, with full compaction, is presented in Figure 6(b). All algorithms except CF perform deallocation in constant time by adding the deallocated memory range to a data structure that keeps track of the free memory slots. CF performs compaction upon deallocation, and therefore takes linear time (in the size of the memory object) for deallocation. The overhead of performing compaction leads to longer execution time, but both CF implementations are bounded and create predictable memory. For the given block-frame size of 32B, CFNM does not perform better than CFM since returning blocks to the free-list of free block-frames takes approximately the same time as moving a whole memory object. Experiments showed that the minimum block-frame size for which deallocation in CFNM is faster than in CFM is 80B.

Using the partial compaction strategy results in constant deallocation times for CFM, as shown in Figure 6(c). Note that this graph shows the same picture as Figure 6(b) except for CFM and CFNM where partial compaction is applied. The compaction bounds for partial compaction are set sufficiently wide to avoid compaction. CFNM shows a step function with tight bounds per size-class. The average number of instructions for deallocation with CFM and partial compaction is 185.91, the standard deviation is 16.58.

5.3 Results: Rate-Monotonic Tests

In the rate-monotonic scheduling benchmarks, we use a set of five periodic tasks resembling a typical scenario found in many real-time applications. Each task allocates memory objects of a given size, and deallocates them when the task's holding time expires. Three of the tasks allocate larger objects and have long holding times, the other two allocate small objects and have short holding times. The tasks have various periods and deadlines. They are scheduled using a rate-monotonic scheduling policy. Since the different tasks create a highly fragmented memory, this benchmark represents a memory fragmentation stress test.

For better readability, the \$ y\$-axes of the graphs show the cumulative number of instructions, i.e., the sum of the number of executed instructions for all operations starting from the first invoked operation up to the currently invoked one. The \$ x\$-axes show the number of invoked operations, as before. Note that a linear function represents memory operations that take constant time.

Figure 7: Rate-monotonic microbenchmark

Image all_malloc_random_instr_sum

(a) Allocation

Image all_free_random_instr_sum

(b) Deallocation with compaction

Image all_free_random_instr_no_compaction_sum

(c) Deallocation, partial compaction

The allocation measurements are presented in Figure 7(a). Best-fit is highly unpredictable. Half-fit and TLSF are constant and fast. CFM is also constant, faster than DL, but slightly slower than Half-fit and TLSF. On average, a CFM allocation request takes 169.61 instructions with a standard deviation of 8.63.

Figure 7(b) shows the deallocation measurements. The differences in growth of the CFM curve correspond to compaction. During the first 550 deallocation operations CFM has to perform a lot of compaction operations but afterwards no compaction is necessary. The total runtime is shorter than the time needed for DL. CFNM takes linear time in the size of the memory object even if there is no compaction performed. The curve reflects this property.

Applying partial compaction leads to constant-time deallocation with CFM and makes it fast and more predictable, as shown in Figure 7(c). This graph shows the same picture as Figure 7(b) except for CFM and CFNM where partial compaction is applied. In order to apply partial compaction we have used the following compaction bounds

on the 46 size-classes in the system: In the size-classes 15-18 and 28-29, two not-full pages are allowed. In the size-classes 19-27, we allow for three not-full pages. All other size-classes can have at most one not-full page. The mean number of instructions for CFM deallocation with partial compaction is 171.61, the standard deviation is 5.09.

5.4 Results: Fragmentation Tests

Our final experiments measure fragmentation. We compare CFM (with partial compaction) with TLSF, since the latter is considered the best existing real-time memory management system in terms of fragmentation [9]. The results are shown in Figure 8. The numbers next to CFM, e.g. CFM 3, denote the maximal number of not-full pages allowed in each size-class.

Figure 8: Fragmentation

Image fragmentation_plot

For the experiments we have used a mutator that allocates the whole memory using memory objects of size 20B-100B. Before we run the fragmentation test around 20% of the number of allocated objects is freed. The memory holes are randomly distributed throughout the memory. The fragmentation tests count how many objects (\$ y\$-axis) of size 20B-16000B (\$ x\$-axis) are still allocatable by each memory management system. CFM obviously deals with fragmentation better than TLSF, even if we allow up to nine not-full pages in each size-class. Moreover, the fragmentation in CFM is fully controlled and predictable.

6 Discussion and Conclusion

Compact-fit is an explicit real-time memory management system that handles fragmentation through compaction like some implicit real-time memory management systems do. Its main new contribution is predictable response times in combination with predictable memory fragmentation.

We have designed and implemented two versions of Compact-fit. Allocating an object takes constant time in the

moving implementation and linear time (in the object's size) in the non-moving implementation. Deallocating an object takes linear time (in its size) in both implementations. If no compaction occurs, deallocating takes constant time in the moving implementation. Dereferencing takes constant time in both implementations.

Hence, we provide tight bounds on the response times of memory operations. Moreover, we keep each size-class (partially) compact, i.e., we have predictable memory. Hence, unlike the other existing real-time memory management systems that do not fully control fragmentation, our compacting real-time memory management system is truly suitable for real-time and even safety-critical applications.

Finally, another real-time characteristic of our memory management system is the constant initialization time. This is achieved using the free-list concept for all resources (abstract addresses, pages, block-frames, etc.) that need initialization.

The experiments validate our asymptotic complexity results. Due to more administrative work related to compaction, our system is slightly slower than the existing systems with real-time response bounds.

There are several possible improvements to our design and implementation that we leave for future work. In our present work, the abstract address space is statically pre-allocated to fit the worst case. For less memory overhead, we could implement a dynamic abstract address space allocation by using the pages from the concrete address space also for storing abstract addresses. Moreover, the present implementation allows for memory objects of size at most 16KB, the size of a page. Arraylets [3] can be used in order to handle objects of larger size. Other topics for future work are concurrency support, program analysis for determining optimal partial compaction bounds and needed amount of abstract addresses, and allocatability analysis.

Acknowledgments

This work is supported by a 2007 IBM Faculty Award, the EU ArtistDesign Network of Excellence on Embedded Systems Design, and the Austrian Science Fund No. P18913-N15.

Bibliography

- 1 BACON, D. F.
Realtime garbage collection.
Queue 5, 1 (2007), 40-49.
- 2 BACON, D. F., CHENG, P., AND RAJAN, V. T.
Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java.
In Proc. LCTES (2003), ACM Press, pp. 81-92.
- 3 BACON, D. F., CHENG, P., AND RAJAN, V. T.
A real-time garbage collector with low overhead and consistent utilization.
In Proc. POPL (2003), ACM Press, pp. 285-298.
- 4 BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R.
Hoard: A scalable memory allocator for multi-threaded applications.
In Proc. ASPLOS (2000), ACM Press, pp. 117-128.
- 5 GUMSTIX.
Gumstix inc.
<http://www.gumstix.org>.
- 6 JONES, R.
The garbage collection page.
<http://www.cs.ukc.ac.uk/people/staff/rej/gc.html>.
The definitive on-line resource for garbage collection material.
- 7 KNUTH, D. E.

Fundamental Algorithms, second ed., vol. 1 of The Art of Computer Programming.
Addison-Wesley, 1973.

- 8 LEA, D.
A memory allocator.
Unix/Mail/, 6/96, 1996.
- 9 MASMANO, M., RIPOLL, I., AND CRESPO, A.
A comparison of memory allocators for real-time applications.
In Proc. JTRES (2006), ACM press, Vol. 177, pp. 68-76.
- 10 MASMANO, M., RIPOLL, I., CRESPO, A., AND REAL, J.
TLSF: A new dynamic memory allocator for real-time systems.
In Proc. ECRTS (2004), IEEE Computer Society, pp. 79-86.
- 11 OGASAWARA, T.
An algorithm with constant execution time for dynamic storage allocation.
In Proc. RTCSA (1995), IEEE Computer Society, pp. 21-27.
- 12 PADALA, P.
Playing with ptrace, part I.
Linux Journal 2002, 103 (2002), 5.
- 13 PUAUT, I.
Real-time performance of dynamic memory allocation algorithms.
In Proc. ECRTS (2002), IEEE Computer Society, pp. 41-49.
- 14 SIEBERT, F.
Eliminating external fragmentation in a non-moving garbage collector for Java.

Decoupling dynamic program analysis from execution in virtual environments

Jim Chow Tal Garfinkel Peter M. Chen

Vmware

Abstract

Analyzing the behavior of running programs has a wide variety of compelling applications, from intrusion detection and prevention to bug discovery. Unfortunately, the high runtime overheads imposed by complex analysis techniques makes their deployment impractical in most settings. We present a virtual machine based architecture called Aftersight ameliorates this, providing a flexible and practical way to run heavyweight analyses on production workloads.

Aftersight decouples analysis from normal execution by logging nondeterministic VM inputs and replaying them on a separate analysis platform. VM output can be gated on the results of an analysis for intrusion prevention or analysis can run at its own pace for intrusion detection and best effort prevention. Logs can also be stored for later analysis offline for bug finding or forensics, allowing analyses that would otherwise be unusable to be

applied ubiquitously. In all cases, multiple analyses can be run in parallel, added on demand, and are guaranteed not to interfere with the running workload.

We present our experience implementing Aftersight as part of the VMware virtual machine platform and using it to develop a realtime intrusion detection and prevention system, as well as an offline system for bug detection, which we used to detect numerous novel and serious bugs in VMware ESX Server, Linux, and Windows applications.

1 Introduction

Dynamic program instrumentation and analysis enables many applications including intrusion detection and prevention [18], bug discovery [11, 26, 24] and profiling [10, 22]. Unfortunately, because these analyses are executed inline with program execution, they can substantially impact system performance, greatly reducing their utility. For example, analyses commonly used for detecting buffer overflows or use of undefined memory routinely incur overheads on the order of 1040x [18, 26], rendering many production workloads unusable. In non-production settings, such as program development or quality assurance, this overhead may dissuade use in longer, more realistic tests. Further, the performance perturbations introduced by these analyses can lead to Heisenberg effects, where the phenomena under observation is changed or lost due to the measurement itself [25].

We describe a system called Aftersight that overcomes these limitations via an approach called decoupled analysis. Decoupled analysis moves analysis off the computer that is executing the main workload by separating execution and analysis into two tasks: recording, where system execution is recorded in full with minimal interference, and analysis, where the log of the execution is replayed and analyzed.

Aftersight is able to record program execution efficiently using virtual machine recording and replay [4, 9, 35]. This technique makes it possible to precisely reconstruct the entire sequence of instructions executed by a virtual machine, while adding only a few percent overhead to the original run [9, 35]. Further, as recording is done at the virtual machine monitor(VMM) level, Aftersight can be used to analyze arbitrary applications and operating systems, without any additional support from operating systems, applications, compilers, etc.

Aftersight supports three usage models: synchronous safety, best-effort safety, and offline analysis. First, for situations where timely analysis results are critical (e.g., intrusion detection and prevention), Aftersight executes the analysis in parallel with the workload, with the output of the workload synchronized with the analysis. This provides synchronous safety that is equivalent to running the analysis inline with the workload. Second, for situations that can tolerate some lag between the analysis and the workload, Aftersight runs the analysis in parallel with the workload, with no synchronization between the output of the workload and the analysis. This best-effort safety allows the workload to run without being slowed by the analysis. Often analyses whose performance impact would be prohibitive if done inline can run with surprisingly minimal lag if run in parallel. Third, Aftersight can run analyses offline for situations where analyses are not known beforehand or are not time critical, such as when debugging.

Aftersight is a general-purpose analysis framework. Any analysis that can run in the critical path of execution can run in Aftersight, as long as that analysis does not change the execution (this would break the determinism that Aftersight's replay mechanism relies upon). Also, Aftersight makes the entire system state at each instruction boundary available for analyses, providing greater generality than approaches based on sampling. Further, logs originating from the VMM can be replayed and analyzed in different execution environments (e.g., a simulator or VMM). This flexibility greatly eases program instrumentation and enables a variety of optimizations.

We have implemented an Aftersight prototype on the x86 architecture, building on the record and replay capability of VMware Workstation. Our framework enables replay on the QEMU whole-system emulator, which supports easy instrumentation during replay and analysis. With this framework, we have implemented an online security analysis that can be used to detect buffer overflow attacks on running systems. We also implemented an analysis that can perform checks for memory safety and heap overflows, and we used this analysis to discover several new and serious bugs in VMware ESX Server, Linux, and Windows applications.

2 The case for decoupled analysis

Aftersight improves dynamic analysis by decoupling the analysis from the main workload, while still providing the analysis with the identical, complete sequence of states from the main workload. This combination of de-coupling

and reproducibility improves dynamic analysis in the following ways.

First, Aftersight allows analyses to be added to a running system without fear of breaking the main workload. Because Aftersight runs analyses on a separate virtual machine from the main workload, new analyses can be added without changing the running application, operating system, or virtual machine monitor of the main workload.

Second, Aftersight offers users several choices along the safety/performance spectrum. Users who can tolerate some lag between the analysis and the workload can improve the performance of the workload and still get best-effort safety or offline analysis, while users who require synchronous safety can synchronize the output of the workload with the analysis.

Third, with best-effort safety or offline analysis, After-sight can improve latency for the main workload by moving the work of analysis off the critical path. Because analyses no longer slow the primary system's responsiveness, heavyweight analyses can now be run on realistic workloads and production systems without fear of perturbing or unduly slowing down those workloads. For example, system administrators can use intensive checks for data consistency, taint propagation, and virus scanning on their production systems. Developers can run intensive analyses for memory safety and invariant checking as part of their normal debugging, or as additional offline checks that augment testing that must already be performed in a quality-assurance department. As an extreme illustration of the type of heavyweight analysis enabled by Aftersight, computer architects can capture the execution of a production system with little overhead, then analyze the captured instruction stream on a timing-accurate, circuit-level simulator. Even when providing synchronous safety, Aftersight can sometimes improve performance compared to running the analysis inline by leveraging the parallel execution of the workload and the analysis.

Fourth, Aftersight increases the parallelism available in the system by providing new ways to use spare cores. Aftersight can run an analysis in parallel with the main workload, and it can run multiple analyses in parallel with each other.

Fifth, Aftersight makes it feasible to run multiple analyses for the exact same workload. Without Aftersight, the typical way to run multiple analyses is to conduct a separate run per analysis, but this suffers from the likelihood of divergent runs and inconsistent analyses. Aftersight, in contrast, guarantees that all analyses operate on the same execution. In addition, each analysis takes place independently, so programmers need not worry about unforeseen interactions between the analyses. Nor must they worry about perturbing the source workload with their analysis. Aftersight allows the number of simultaneous analyses to scale with the number of spare processors in a system, all while not affecting the performance of the primary system.

Sixth, Aftersight makes it possible to conduct an analysis that was not foreseen during the original run. This ex post facto style of analysis is particularly powerful when it is difficult to anticipate exactly what must be analyzed. For example, analyzing computer intrusions invariably requires one to examine in detail a scenario that was not foreseen (else, one would have prevented the intrusion). Debugging performance or configuration problems leads to a similar need for conducting unforeseen analysis. Aftersight allows the user to iteratively develop and run new analyses, all on the same exact execution.

Finally, by decoupling analysis from the main execution, Aftersight allows the analysis and execution components to be individually optimized to their intended function. The main workload execution can be performed on a platform optimized for realtime performance and responsiveness (such as a commercial VMM), while analysis can be delegated to a platform optimized for ease of instrumentation (such as an extensible simulator).

Figure 1: Overview of current system.

3 Architectural overview

Our current Aftersight system targets the x86 architecture and has three main components: the virtual machine monitor (VMM), deterministic VM record/replay, and an analysis and instrumentation framework. For our prototype, each of these pieces builds on functionality of existing off-the-shelf components. In this section we examine aspects of these components that are relevant for decoupled analysis. In Section 5 we look at how they

can be modified and integrated to facilitate decoupled analysis.

3.1 VMM

A VMM provides an environment for running arbitrary guest operating systems and applications in a software abstraction of the hardware [23]. No modifications to a standard VMM are required to support decoupled analysis, except for the need to support replay. Aftersight uses VMware Workstation, a highly optimized production x86 VMM. VMware Workstation uses a hosted architecture [28], i.e. it uses a host operating system to access physical devices like disk or network.

As Aftersight is based on a virtual machine platform, it inherits a variety of useful and desirable traits. First, any individual process in the guest VM as well as the guest OS kernel itself can be a target of Aftersight instrumentation and analysis. Furthermore, a range of target systems are supported without extra work: a single implementation works across OSes, processes, languages, etc.

Next, virtualization is becoming increasingly ubiquitous in a wide range of computing environments. Because Aftersight can be provided as a service of the virtual infrastructure with nominal overhead, there is a relatively easy path to adoption. With such a primitive in place, the deployment of new monitoring and analysis tools can be a continuous, normal part of the execution of guests.

Finally, operating at the VMM level gives Aftersight visibility at all layers of the software stack. It can be used to analyze operating systems, applications, and interactions across components. This generality is critical for applications ranging from performance analysis, to tracking the flow of sensitive or potentially malicious data in a system.

3.2 Deterministic VM record and replay

Aftersight builds upon the replay facilities in VMware Workstation 1. A deterministic VM record/replay system records enough information about a running workload to reproduce its exact instruction sequence. To support replay, a VM must record and replay all inputs to the CPU that are not included in the state of the guest memory, registers, or disk. This includes reads from external devices, such as the network, keyboard, or timer,

and asynchronous events such as interrupts. Recording these nondeterministic inputs enable VM replay to recreate the whole instruction stream [4, 9, 35]. As with other software-based replay systems [9], VMware Workstation is not able to replay virtual multiprocessors.

VM replay systems are highly efficient in time and space overhead [9, 35]. This efficiency comes about because nearly all instructions produce the same result given the same inputs, and most instructions use only the results of previous instructions as their inputs. Because of this domino effect, a long sequence of instructions can be exactly reproduced while only supplying a few values that come “from outside” the system.

A study [35] of VMware’s then-current replay implementation showed performance overheads for SPEC benchmarks as low as 0.7% and an average of 5% [35]. Another replay implementation for the x86 [9] reported similar overheads. Overheads will generally be workload dependent, however. Worst-case performance observed in [35] reached 31% and 2.6x for some workloads. However, many of the chief bottlenecks were not fundamental [35], and subsequent improvements have lowered these overheads.

Trading the overhead of analysis for the overhead of VM replay is a compelling exchange for many heavyweight analyses. However, even for lightweight analyses, the ability to run multiple or ex post facto analyses still provides reason to use decoupled analysis.

While Aftersight uses VM replay, replay can also be implemented at many levels besides the VM-level, such as the OS process-level [27], the JVM-level [8], or the disk level [32]. VM replay stands out for a couple reasons.

First, a single VM replay implementation enables one to replay the entire state of all software on that hardware platform regardless of operating system, language or runtime environment. In contrast, other types of replay can only work for a small subset of available software as they are heavily dependent on the particular language and operating system they are designed for. Reimplementing replay for each OS or language variant would be a herculean task.

Second, a VM replay system often has lower overhead than higher-level recording. For example, one could replay

the file system at the system-call level, but this would require all file system calls to be recorded, including every read of every file. However, a VM-based system need only record nondeterministic VM inputs, and this frees it from recording reads from the file cache or disk. Similarly, a process-level record/replay system must record reads from IPC pipes, files, and all system calls that return data, while these can all be ignored by a VM-based solution.

3.3 Analysis framework

Aftersight does instrumentation and analysis dynamically during replayed execution. Normally in a VM record/replay system, the same VMM is used during both recording and replaying. A key property of Aftersight is its ability to support heterogenous replay, i.e. the ability to use one platform to execute and record a workload and a different platform to replay and analyze, with each platform tuned for its particular purpose.

The Aftersight prototype relies on a VMM for execution and recording, whereas replay and analysis can be done in a VMM or a simulator. A VMM is an excellent platform for recording, because it is optimized to minimize recording overhead to support production environments. However, platforms such as software simulators are often better suited to supporting general-purpose analysis.

For example, many VMM environments don't provide a simple, low overhead way to instrument every memory access. This can be implemented on top of page protections and faulting on every memory access. However, a software simulator can often accomplish this task faster and with less effort than a VMM will natively.

Analysis environments Dynamic instrumentation can be implemented in many ways. Most simply, we can build ad-hoc hooks into our replaying environment that supply callbacks when events of interest happen.

In our Aftersight prototype, we implement dynamic instrumentation through dynamic binary translation (BT). BT is the technique of dynamically translating a set of instructions into an alternate set of instructions on-the-fly, which are then executed. Techniques such as caching translations [33] can be used to make this process very efficient. Affecting what translations are produced allows one to very flexibly instrument a running program.

Our prototype offers two BT environments to analysis applications: one is based on VMware Workstation, the other on QEMU [3], which is an open-source x86 simulator. Both offer the ability to run code using BT alone, or in some combination with native execution [1]. However, each has its own strengths and limitations.

VMware's BT is extremely fast, but it is optimized for performance rather than extensibility. For example, VMware's BT does not support an extensible intermediate representation (IR). An extensible IR is commonly used in general purpose BT systems [17, 3] to abstract the x86's CISC-style instructions into a more instrumentation-friendly RISC-style format. However, these additional translation costs make little sense given VMware's specialized use of BT. Also, for efficiency reasons, VMware BT runs in ring 0, and in an environment where dynamic memory allocation is heavily constrained. Developing general-purpose analyses under these constraints is quite burdensome, and the resulting analyses may even be slower from having to work with limited memory.

In contrast, QEMU is not nearly as fast as VMware Workstation, but it is much more flexible: it provides an extensible IR and runs as a regular user-mode process, which means normal program facilities like malloc, gdb, etc. are available. In converting it to enable replay, we stripped out much of its now unnecessary functionality, to the point where it is little more than a simple CPU simulator. The virtual device model, including the disk, network, the chipset, and the local APIC, have all been removed. All that remains are the components needed to deal with instruction execution and memory access.

Of course, because analysis is decoupled, other special-purpose analysis environments could be built to better suit the needs of particular analyses if desired.

4 Online analysis

In two of Aftersight's three usage models, synchronous safety and best-effort safety, the analysis runs in parallel with the workload. Aftersight makes it easy to simultaneously record and analyze a workload. In our prototype, recording generates a replay log on disk. Analysis VMs can run on separate cores and process the log as it is

being generated by the primary VM. Analysis can even take place across multiple machines by reading the log file over the network, since network bandwidth is more than adequate for most workloads. Log sizes are often quite modest [9, 35]. For example, Xu et al. [35] notes that only 776 KB of compressed log space was necessary to record an entire Windows XP bootup–shutdown sequence.

This section describes how Aftersight synchronizes the main workload with the analysis when the two are running in parallel, and how running the analysis in parallel with the workload can speed up the analysis.

4.1 Synchronization

When running in simultaneous record and analysis mode, analysis results may affect the operation of the primary VM (e.g., a security check may detect an intrusion and halt the system). When this feedback is needed, Aftersight can take one of two strategies to synchronize the execution of the primary and analysis VMs.

The need for synchronization arises because the primary VM executes ahead of the analysis VM. The portion of the primary VM's execution that has not yet been run on the analysis VM is speculative. This speculative portion will usually be committed by the analysis VM as checks complete. In the rare case when checks fail, the speculative portion of execution differs from what would have been executed with inline analysis.

The first method for synchronizing provides synchronous safety, which is equivalent to running the analysis inline with the workload. To provide this guarantee, Aftersight defers the output of the primary VM (e.g., network packets) while they are speculative, i.e., until the analysis reaches the same point in execution. Deferring outputs while they are speculative ensures that the released outputs of the primary VM are identical to those of a system with inline checks, even though the internal state of the primary VM may differ from a system with inline checks [19].

In addition to synchronizing the primary's output with the analysis VM, we could also limit how far the primary is allowed to run ahead of the analysis VM. Limiting the lag between primary and backup limits the amount of time that the primary's outputs are deferred, which in turn limits the amount of timing perturbation the primary VM may observe (e.g., when it measures the round-trip time of a network).

Deferring output in the above manner provides the same safety guarantee as if the analysis were running inline with the workload. However, it may hurt performance by blocking the output of the primary VM.

A different point in the safety/performance spectrum optimizes performance but relaxes the safety guarantee by giving lazy feedback to the primary VM. In this case, the main workload executes at full speed and is not slowed by the work of analysis. Rather, analysis results are fed back to the main workload as they become available. This usage model is useful when the analysis is too heavyweight to run with stronger safety guarantees, or when the analysis does not require such guarantees, as in profiling or debugging.

4.2 Accelerating analysis

The analysis VM in Aftersight executes the same instructions as the primary VM, and it also does the work of analysis. Because the analysis VM is doing more work than the primary VM, it can easily become a bottleneck, especially when providing synchronous safety. This section describes several ways to improve the performance of the analysis VM, to allow it to better keep up with the primary VM.

First, a surprising amount of performance can be won from a basic aspect of replayed VM execution: interrupt delivery is immediate. x86 operating systems use the hlt instruction to wait for interrupts; this saves power compared to idle spinning. During analysis, hlt time passes instantaneously. One hlt invocation waiting for a 10ms timer interrupt can consume equal time to tens of millions of instructions on modern 1+GHz processors. Section 6.3 provides more detail on the boost this can have on performance.

Second, device I/O can be accelerated during replay. For example, network writes need not be sent, and network reads can use data from the replay log. This frees guests from waiting for network round-trip times, especially because disk throughput is often greater than end-to-end network throughput. Disk reads can similarly be satisfied from the replay log rather than from the analysis VM's disk, and this can accelerate the analysis VM because the replay log is always read sequentially. This optimization can also free the analysis VM from executing disk writes during replay, which frees up physical disk bandwidth and allows write completion interrupts to be

delivered as soon as the guest arrives at an appropriate spot to receive them. Disk reads done by the primary VM may also prefetch data and thereby accelerate subsequent reads by the analysis VM [5].

Third, a number of opportunities allow Aftersight to memoize operations that happen during record that don't need to be fully replayed. An example of this is exception checking.

There are many times where the x86 needs to check for exceptional conditions. Although these checks rarely raise exceptions, executing them adds considerable overhead in our CPU simulator. Segment limit checks are an example: every memory reference or instruction fetch must be checked that it is within bounds for an appropriate segment.²

Decoupled analysis allows one to reduce the overhead of exception checking on the analysis VM by leveraging the exception checking that has already occurred on the main VM. The time and location in the instruction stream of any exceptions are recorded by the main VM, and these exceptions are delivered during replay just like asynchronous replay events. This strategy frees the analysis VM from the overhead of explicitly checking for exceptions during replay. Memoizing these checks makes the CPU simulator faster and less complex, while still guaranteeing proper replay of a workload that contains violations of the checks.

There are many x86 checks that can be memoized, although we have not yet implemented this optimization in Aftersight: debug exceptions, control transfer checks for segment changes, the alignment check (which when enabled, ensures all memory accesses are performed through pointers aligned to appropriate boundaries), and others.

5 Implementation and integration

While Aftersight builds on existing components, leveraging these for decoupled analysis poses a variety of challenges. This section discusses how to adapt a simulation environment to replay VMM logs and the challenges posed by the heterogeneous combination of record and replay components.

Aftersight uses different platforms for recording and analysis. For recording, Aftersight uses VMware Workstation, which is designed to minimize the time and space overhead of recording. For analysis, Aftersight can use VMware Workstation or QEMU. Simple analyses can be conducted by modifying VMware Workstation's BT, while more general analyses are easiest to implement in QEMU, which is designed for flexibility rather than pure speed.

For replay and analysis, compatibility is an issue for both platforms. When VMware Workstation replays a log, no compatibility issues arise with devices, chipset, etc. because the emulation code is identical. However, because it relies directly on the hardware for CPU emulation, replay is generally infeasible if the processor is significantly different (e.g., attempting to replay a log from an Intel CPU on an AMD platform). In contrast, with a CPU simulator like QEMU we can easily support a wide range of CPU families on a single hardware platform. However, QEMU does not have the same device models as the recording platform. In this next section, we look at how Aftersight bridges the compatibility gap between the VMware Workstation recording and QEMU in two areas: I/O device emulation and hardware performance counters.

5.1 Device emulation

The first gap between our recording platform (VMware Workstation) and one of our analysis platforms (QEMU) is device emulation. QEMU emulates different I/O devices than VMware, which prevents QEMU from directly consuming the log recorded by VMware.

To understand the problem and the solution we adopted, it is helpful to consider two different methods for recording device interactions. The first method is to record all outputs from an emulated device to the CPU. During replay, the recorded values would be resupplied to the CPU (presumably the guest OS device drivers). This method is ideal for compatibility between the recording and analysis platform because no device emulation is needed during replay.

However, VMware Workstation and other VM replay systems [9] use a second method to record and replay device interactions. Instead of recording the output from the emulated devices, they record the nondeterministic,

external inputs to those devices. During replay, these recorded inputs are redelivered to the devices, and these allow the emulated devices to be deterministically replayed along with the CPU.

VM replay systems use this second method for two reasons. A main reason is that the second method allows a replaying session to “go live” —to stop replaying and start responding to new input—at any point while replaying. In contrast, recording and replaying the outputs of the emulated device without replaying the emulated device itself means that the emulated device is not available to go live. Another reason is that it can drastically reduce the amount of data that must be recorded. For example, to replay a disk read operation, the first method must record the actual data being read from the emulated disk, while the second method need only record the nondeterministic inputs to the disk (note that the inputs from the CPU to the disk are deterministic and need not be recorded).

Unfortunately, recording only the nondeterministic inputs to the device leads to a compatibility problem during analysis. Whereas the VM recording system assumes that the replaying system can replay the emulated device, QEMU and other flexible analysis systems usually will not emulate the exact same devices used during recording.

Aftersight bridges the compatibility gap between recording and analysis for devices by adding a relogging step to replay. We modified VMware’s replay system to record a new log during replay, which contains all outputs from the emulated device to the CPU, including responses to I/O requests, interrupt delivery, and effects on memory. This log is equivalent to one generated by the first method of replaying devices and has the same compatibility advantages, i.e. the analysis system needs no device emulation during replay.

While our modified VMware VMM supports relogging, none of our modifications to support relogging impact the record side operation of the VMM, since relogging is only active during replay.

5.2 Hardware performance counters

The second gap between our recording platform (VMware Workstation) and one of our analysis platforms (QEMU) is hardware performance counters. VM replay implementations will normally use hardware counters to determine when a nondeterministic event happens during recording, as well as to trigger that event during replay [4, 9].

These counters record aspects of the dynamic instruction stream that help to uniquely position an event in time such as the instruction count [4], or the number of branches executed [9].

Instructions added dynamically by BT, as well as by analysis instrumentation, disrupt counts kept by the hardware by adding dynamic instructions in an unpredictable manner. This makes hardware counters difficult to use directly by our CPU simulator.

Instead, our CPU simulator emulates the accounting provided by the hardware counters in the translations it emits. These translations include a small amount of code to update counts and dispatch to an event handler when it is time to deliver an asynchronous nondeterministic event (such as an interrupt or DMA).

QEMU doesn't normally allow interrupt delivery within a basic block [2] of instructions. Instead, these events are delayed until the current basic block completes. The VMware recording system contains no such artificial restriction, so we needed to remove this restriction of QEMU to replay VMware's log. When our stripped-down QEMU reaches a basic block containing a replay event, it will emit new translations for the block. The block is split into two halves: the block of instructions before the replay event, and the block after. Checks between basic blocks will determine that the BT system can deliver the event.

6 Evaluation

Aftersight makes it possible to run heavyweight analyses on realistic workloads with several options along the safety/performance spectrum. In this section, we evaluate the performance of Aftersight under three usage models. We first show how Aftersight can provide synchronous safety with slightly higher performance than a system using inline analysis. Next, we show how Aftersight with best-effort safety makes it possible to run heavyweight analyses in parallel with the main workload and how the techniques described in Section 4.2 allow heavyweight analyses to keep up with the main workload. Last, we demonstrate the utility of enabling heavyweight analyses that are built and applied after the main workload completes.

Figure 2: Aftersight performance with synchronous safety.

6.1 Synchronous safety

We first evaluate how Aftersight performs when providing safety that is equivalent to running the analysis inline with the main workload. In this usage model, Aftersight runs the analysis in parallel with the main workload, and defers the output of the workload until the analysis reaches that output. Aftersight's main benefits for this usage model are the ability to add new analyses without fear of breaking the workload and the ability to conduct later analyses that were not envisioned at the time of the run.

When providing synchronous safety, Aftersight's performance is limited by the analysis VM. A reasonable expectation is that the performance of the analysis VM will be comparable to that of an inline system (or slower, due to replaying overhead). While this can be true for many workloads, the analysis VM in Aftersight can also run faster than an inline system by taking advantage of the work done by the primary VM (Section 4.2). We demonstrate an example of this phenomenon through the following experiment.

We evaluate Aftersight with synchronous safety on a workload that uses wget to fetch a directory of linked web pages from a local httpd web server. The directory of web pages consists of 5000 HTML files, each 200 KB. The workload starts with a cold file cache and spends most of its time fetching data from disk. The check running in the analysis VM mimics a trivial on-access virus scanner by computing for 2 ms on each disk request.

Figure 2 compares the performance for Aftersight with synchronous safety with running the analysis inline with the workload. The analysis VM in Aftersight always trails the primary VM that it is replaying, so the workload is considered complete when the analysis VM completes.

Although the analysis VM is slowed by the overhead of replaying, it leverages the disk reads performed by the primary VM to regain this performance. The net effect on this benchmark is that Aftersight achieves slightly better performance than inline analysis.

6.2 Best-effort safety

We next demonstrate how Aftersight enables heavyweight analyses to execute concurrently with a workload with best-effort safety. Our analysis enforces protection for guest address spaces at the granularity of individual bytes of memory. This supports checking for a wide range of memory errors, though we only apply it to catching heap overflows in our example.

An in-memory bitmap specifies whether each byte of a particular address space is writable or not. The bitmap is organized as a two-level page table to conserve space. To implement the checks, the analysis dynamically instruments instructions that write to memory. These writes are translated to look up the appropriate protection bits in the bitmap and check if they allow writing. If they do, the write proceeds normally, otherwise the analysis invokes an error handler. When running the analysis in parallel with the main workload (online), the error handler can invoke a feedback action that takes corrective measures. For example, the system can automatically suspend the primary VM when an error is detected.

We implemented the analysis in VMware Workstation by modifying the binary translation done during replay. On each write, the translation saves two scratch registers and the CPU flags, checks the protection bits in a bitmap, then restores the two scratch registers.

To use this analysis, the guest kernel specifies the desired protection for each byte of memory of an address space. We modified a guest Linux kernel to use this facility to do heap-overflow bounds checks on dynamically allocated kernel objects. Linux already includes a facility for adding “red zone” buffers to the start and end of every `kmallocobject`. We use this facility and add code to set the bitmap permission bits for these redzones appropriately. Only kernel code needs checking, since normal page protections prevent user code from touching kernel heap objects.

Without Aftersight, this analysis is too slow to be used in production settings. We measured the speed of analysis without Aftersight by running the checks inline in the VMware binary translator of the main workload. The results are shown in Figure 3. For a kernel compilation benchmark, running the benchmark with the analysis inline (i.e.

without Aftersight) takes 191.10 seconds (a 2.48x slowdown compared to running the benchmark without the analysis). Running the workload with Aftersight reduces the time to complete the benchmark to 84.86 seconds. With best-effort safety, Aftersight reduces the perturbation on the main workload by moving the overhead of analysis off the critical path of the main workload and into a separate analysis system. The main workload pays for this in the form of record/replay overhead, which for this benchmark is 10.4%.

Figure 3: Comparison of kernel compile overhead with heap buffer overflow detection: Aftersight runs a kernel compilation benchmark nearly 2x faster than inline checking. Checks are still run concurrently with variable lag.

A potential disadvantage of decoupling this analysis from the main workload is that the detection of a write violation may occur long after the offending instruction execution. However, even delayed feedback can be very useful. For example, we ran an SSH server in the primary VM, logged into it from an outside client, and invoked a system call that contains an erroneous heap overflow. As before, we check byte-level write protections in an analysis VM. We measured how long it takes the analysis VM to discover the problem, assuming that prior to the SSH connection, both the primary and analysis VMs are synchronized to the same point in time.

In Figure 4 we see the progress of the primary and analysis VM, measured in #branches executed vs. wall clock time. As shown by the horizontal distance on the graph between identical branch counts in the primary and analysis VM, the analysis VM lags the primary VM by varying amounts during the run. In this experiment, there was a 0.86 second latency between the write violation in the primary VM and its detection in the analysis VM.

Decoupled analysis can lead to delays between an event in the main VM and the analysis of those events in the analysis VM. This delay is not a problem for analyses that don't need to provide feedback to the main VM, or for analyses that have no response time requirement on their feedback (such as optimization or bug finding). The delay is not ideal for analyses that implement security or correctness checks. However, delayed feedback can still be very useful, and is certainly preferable to being unable to run heavyweight analyses at all.

Figure 4: Measuring latency of simultaneous record and analysis for write protection violations: The two dotted

lines at the end show when the write violation occurs and when it is identified by the analysis (shown separated by 0.86 seconds).

6.3 Idle-time boost

Section 6.2 showed that decoupling analysis from the main workload with best-effort safety can improve the performance of the main workload, but that the analysis may complete significantly after the workload. Since heavyweight analyses may be much slower than the main workload, a natural question is “How can heavyweight analyses keep up with the main workload over a long run?”. Section 4.2 describes some techniques that help a replaying analysis VM to keep up with a main VM, and this section evaluates the effectiveness of these techniques. In particular, we observe that idle time in the primary can provide enough time for the analysis VM to catch up during replay, even for very heavyweight analyses.

As an example of a heavyweight analysis platform, we configure the VMware Workstation binary translator to translate all guest instructions (instead of just guest kernel instructions), but still with a relatively small (2.9 MB) code cache. Even with no extra analysis, this configuration of VMware Workstation runs several times slower than the normal configuration.

We measured the overhead of this analysis system on a CPU-bound workload, which is winLAME (a GUI frontend to the LAME mp3-encoder) in a Windows XP VM, encoding .wav files into mp3 format. The test machine is a two-processor, dual core 2GHz AMD Opteron (containing four logical CPUs) running Debian Linux. On this workload, the analysis VM takes 4.65x as long to complete the workload as the main VM, due to the overhead of the slower binary translator,

Most realistic workloads can be replayed much faster by skipping over idle time in the main workload. To see how idle time can provide a boost, we ran this analysis again with an interactive, desktop workload. In the primary VM, we use Windows XP and:

- * Start Firefox. Edit the proxy settings to get out of the corporate network.

- * Visit slashdot.org, scroll through the front page, browsing for one minute.
- * Visit internal website and download an Excel spreadsheet containing numbers used in this paper.
- * Close Firefox. Start Excel, open the spreadsheet.
- * Create a chart, and plot two curves using data in the spreadsheet. Add a trend line to one of the curves.
- * Close Excel. Open Powerpoint, and create a custom animation using four block arrows flying in from different directions. Close Powerpoint.

Figure 5 shows the results of this experiment. Figure 5(a) shows the progress of the primary VM and the analysis VM as wall clock time progresses. Horizontal gaps between the two curves occur where bursts of high CPU utilization cause analysis to lag the primary.

Figure 5(b) more clearly illustrates these bursts by showing the instantaneous compute rates of the primary and the analysis. Figure 5(b) shows that the primary contains many compute spikes. Meanwhile, the analysis runs at a more constant pace because it is limited by the speed of binary translation. These compute spikes can cause significant lag (one spike causes the main VM to execute 6x faster than the analysis VM). However, as shown in Figure 5(a), the idle times in the workload allows the analysis VM to eventual catch up from this lag.

Figure 5(c) graphs the lag between the main VM and the analysis VM as the workload progresses. This graph demonstrates two major benefits of decoupled analysis with best-effort safety. First, note that the analysis VM can lag behind the main VM significantly (10–11 seconds on average, and as high as 35 seconds behind). These periods of high lag imply that running the analysis inline with the main VM or with synchronous safety would cripple the interactive performance of the main VM (imagine waiting 35 seconds between clicking a button and waiting for the corresponding menu to appear!). In contrast, Aftersight with best-effort safety decouples the analysis, so that users of the primary VM don't experience this lag. Instead, the primary's responsiveness is completely independent of the speed of analysis, e.g. button clicks and menu selections occur at full speed.

Figure 5: Plotting the progress of a source workload, and a online tandem analysis of that workload running on another core: a desktop interactive session shows how analysis (running in a binary translator that is several times

slower) is allowed to lag during bursty periods of computing, and catch up when bursts end.

Second, this graph illustrates how idle times between bursts of CPU activity enable the analysis VM to catch up with the main VM. Although the lag gets as long as 35 seconds during this workload, the analysis VM is able to catch up by the end of the workload. Catching up in this manner is only possible because of decoupled analysis. Synchronizing the main VM with the analysis VM would limit the speed of the main VM during bursts of CPU activity to that of the analysis VM, and it would limit the speed of the analysis VM during idle periods to that of the (idling) main VM.

Idle time in real-world workloads is quite common. In an informal poll, idle time was over 95% for several desktop computers used by full-time computer programmers for compiling programs, editing tex, e-mail, web browsing, and running VMs. Idle time was over 75% for a production web server and mail server at the EECS department of a large public university.

Idle time can also be deliberately increased in many systems, and this may help heavyweight analyses keep up with the main VM. For example, idle time can be increased in server farms by adding more servers and balancing load across them.

6.4 Offline analysis

This section demonstrates how Aftersight enables heavyweight analyses to be built and applied after the main workload completes.

To illustrate this capability, we implemented an analysis that ensures every instruction executed by the source workload meets a set of memory safety guarantees: that a dereferenced pointer must point to valid stack or heap data, that any bit of stack or heap data used in control flow or as a pointer/index must be initialized before use, and that there are no memory leaks (roughly, that the guest executes “Valgrind-safe” [26]).

Asserting continual satisfaction of these constraints is a very expensive job—properly implementing a check for

uninitialized data requires a full taint propagation analysis [26]. Our tool implements this taint analysis and does it at bit-level precision, largely following the implementation given by [26]. The analysis tracks the state of each bit in all guest memory and registers according to the state machine shown in Figure 6. To initialize the state of each bit, the checker interposes on all memory allocation requests for the heap (through calls to memory allocator functions) and stack (through manipulations of the stack pointer). To maintain the value of the state of each bit, the checker interposes on every instruction executed by the workload, for example to propagate the state of source memory or registers to destination memory or registers. The tool uses symbol information to identify calls to the appropriate heap allocator for the system. When analyzing a particular user process, the target process first identifies itself by making a hypercall.

The analysis is built using Aftersight's QEMU-based CPU simulator. The analysis is extremely heavyweight (on the order of 100x) and is best suited to running offline.

Implementing this analysis in Aftersight yields two important benefits relative to traditional tools such as Valgrind and Purify. First, Valgrind and Purify slow their target program too much to be used for long-running, realistic workloads, and they may perturb their target programs too much to capture realistic interactive workloads. In contrast, Aftersight allows long-running, interactive workloads to run with little overhead by allowing the work of analysis to be run later. Second, tools such as Valgrind and Purify can only be applied to user-level code. They cannot be applied to OS or VMM code, even though such code is critical to reliability and safety. In contrast, analyses implemented in Aftersight can be applied to all software running in the virtual machine, including the operating system or even another VMM running inside VMware Workstation.

Figure 6: Memory consistency checker: a state machine is associated with every bit of guest memory and registers, and finds uses of garbage data and dangling pointers.

We have used this tool to find serious bugs in large, complex systems, including kernel code such as VMware ESX Server and Linux. The rest of this section describes bugs we found with this analysis.

ESX Server We used our analysis tool in the development of VMware ESX Server [30] by running ESX inside

a VM hosted by VMware Workstation. We found 10 type safety errors, over half of which were classified as critical or show-stopper bugs, and were able to fix them during development.

For example, in one bug the ESX Server kernel has a utility data structure for recording statistics whose use is sprinkled throughout the code. It takes an array of values as argument, and stores it:

```
Histogram_New(..., const uint32 numBuckets,
```

```
const Histogram_Datatype*
```

```
const bucketLimits) {
```

```
...
```

```
histo = Heap_Alloc(heap,
```

```
sizeof(struct Histogram) +
```

```
bucketCountsSz);
```

```
if (histo != NULL) {
```

```
histo->numBuckets = numBuckets;
```

```
...
```

```
histo->limits.arbitrary.bucketLimits =
```

```
bucketLimits;
```

```
}
```

```
...
```

This array of values is used when manipulations to the statistics occur. Unfortunately, the structure is allocated on the heap, and some callers initialize it with an array from the stack:

```
SCSIAllocStats(Heap_ID heap,
```

```
ScsiStats *stats) {
```

```
Histogram_Datatype limits[...];
```

```
...
```

```
stats->cmdSizeHisto =
```

```
Histogram_New(..., limits);
```

```
...
```

Under certain circumstances, this would cause a crash, but the tool was able to diagnose the problem without reproducing the crash by noticing that using the data structure caused references to data located in popped off stack frames.

Linux We also applied our analysis tool to the Linux kernel by running it as the guest kernel in a VM. Our tool

diagnosed a long-overlooked type safety error in an old part of the core Linux kernel. Its UDP stack makes use of uninitialized stack garbage on reception of UDP packets through `recvfrom`. Whenever `recvfrom` is called, a `msg` structure containing a field `msg_flags` would be allocated on the stack, but never initialized:

```
/* from net/socket.c */
```

```
asmlinkage long sys_recvfrom(...) {
```

```
...
```

```
struct msghdr msg;
```

```
...
```

```
msg.msg_control = NULL;
```

```
msg.msg_controllen = 0;
```

```
msg.msg_iovlen = 1;
```

```
msg.msg_iov = &iov;
```

```
iov.iov_len = size;
```

```
iov.iov_base = ubuf;
```

```
msg.msg_name = address;
```

```
msg.msg_namelen = MAX_SOCKET_ADDR;  
  
if (sock->file->f_flags & O_NONBLOCK)  
  
flags |= MSG_DONTWAIT;  
  
err = sock_recvmsg(sock, &msg, size, flags);  
  
...
```

Following the call stack down through sock_recvmsg, this structure is passed to udp_recvmsg, which uses msg_flags:

Backtrace:

```
#0 udp_recvmsg  
  
(linux-2.6.20.1/net/ipv4/udp.c:843)  
  
#1 sock_common_recvmsg  
  
(linux-2.6.20.1/net/core/sock.c:1617)  
  
#2 sock_recvmsg  
  
(linux-2.6.20.1/net/socket.c:630)  
  
#3 sys_recvfrom
```

```
(linux-2.6.20.1/net/socket.c:1608)
```

```
#4 sys_socketcall
```

```
(linux-2.6.20.1/net/socket.c:2007)
```

```
#5 syscall_call
```

```
(linux-2.6.20.1/arch/i386/kernel/entry.S:0)
```

```
/* net/ipv4/udp.c */
```

```
int udp_recvmsg(..., struct msghdr *msg, ...)
```

```
{
```

```
...
```

```
if (! (&& msg->msg_flags&MSG_TRUNC)) {
```

```
...
```

In this case, the test on `msg_flags` gated a checksum computation, so although no crash would result from this erroneous use of `msg_flags`, it would cause random, unnecessary extra computation to occur on reception of UDP packets.

We discovered this bug in Linux 2.6.20.1 and reported it to kernel developers, who fixed it in the next major

release. This bug existed in the code for many years (all prior versions of 2.6 and all versions of 2.4 we checked back to 2002).

Putty We also tested a common Windows SSH client called Putty. Our tool found one memory leak that was invoked whenever a menu item was selected. We are currently investigating other user programs as well.

7 Future work

There are many interesting open questions about how to optimize and synchronize record and analysis.

Workload memoization Memoizing state generated by native hardware during record can avoid the need for re-computation during analysis, and this can be used to accelerate analysis. We use this kind of memoization when simulating SMM (system management) mode [13]. Our CPU simulator does not implement x86's SMM mode because the version of QEMU we started with did not support it. However, the VMware VMM is more faithful about this part of the architecture, and some target workloads do contain execution in SMM. Aftersight memoizes SMM to maintain compatibility for these workloads. In its relogging step, Aftersight records the changes to memory made in SMM (some of which may be used outside SMM, for example by the guest BIOS code). During replay, the analysis infrastructure reproduces these effects at the proper time, thus avoiding the need to simulate SMM code.

In addition to helping compatibility, memoization can also be used to accelerate replay. For example, consider an analysis where we are only interested in the execution of a specific user process. With memoization, we could use relogging to summarize the execution of all other code in the system into their effects on memory and registers. In essence, this turns the execution of the OS and other processes into the equivalent of a single DMA operation. This would allow subsequent replay and analyses to ignore writes to pages not mapped into the current process, context switches to other processes, and most kernel activity. Focusing on one process would also allow us to accelerate the simulator by not emulating the hardware MMU. Instead we could simply use mmap or its equivalent to set up the address space for the process and allow memory accesses to run natively.

Feedback modes Simultaneous record and analysis mode can use different types of feedback loops to synchronize. We discussed the tradeoffs between two basic approaches, blocking and lazy feedback modes, in Section 4. However, one limitation of these approaches is that they fail to take into account the semantics of the OS, application, or analysis.

If we add more intelligence to our synchronization strategy we can take advantage of natural join points that occur. For example, when analyzing a web server for security, we can impose a synchronization restriction that the analysis VM must be in-synch with the primary whenever the primary initiates an outgoing TCP connection (delaying the primary, if necessary, to guarantee the synchronization), assuming we expect such events to be important but relatively rare. Synchronizing on such an event provides a hard guarantee that can prevent the spread of an attack, yet maximizes the amount of time the analysis machine has to catch up with the primary.

8 Related work

Replay facilities in a VMM have been discussed by a number of researchers [4, 9, 35] and used for a variety of purposes. For example, Bressoud and Schneider log non-determinism to support re-execution of a whole machine (OSes and applications) and use this to tolerate fail-stop faults on HP PA-RISC [4]. ReVirt [9] uses VM replay on x86 systems to enable ex post facto analysis for computer forensics. Aftersight uses VM replay for another purpose, which is to enable heavyweight dynamic analysis to be used on realistic workloads without perturbing them. Aftersight is also more flexible than these past VM replay systems because it allows analyses to run in a different environment from the primary, such as a simulator. Aftersight also leverages the fact that replicas can run faster than the primary to make online analysis more practical.

Researchers have suggested using replay implemented at the virtual-machine level or in hardware to conduct various types of offline analyses, such as computer forensics [9, 14], debugging [15, 16, 34], and architectural simulation [35]. Aftersight makes more types of analysis practical by allowing the analysis to run in a simulator, which reduces the cost of context switching between the replaying virtual machine and the analysis code. Analyses like taint analysis, which require frequent switches, are impractical without this capability. Aftersight also extends the use of VM replay to both online and offline analysis.

Other researchers have sought to run analyses online and in parallel with the original program via software or hardware support. Patil and Fischer proposed running an instrumented “shadow process” in parallel with the original program [21] and used this approach to implement memory safety checks. Speck [19] and SuperPin [31] fork multiple analysis processes from an uninstrumented process, using record/replay to synchronize the analysis processes. Whereas these past approaches can only analyze an application process, Aftersight expands the scope of decoupled analysis to include the operating system and all applications running on a machine. Aftersight also uses a more complete replay system that handles asynchronous interrupts.

Oplinger and Lam leverage proposed hardware support for thread-level speculation (TLS) to enable the original program to run in parallel with monitoring code [20]. They depend on proposed hardware support for TLS to detect data dependencies and rollback the original program if it causes a conflict. Similarly, Zhou, et al. use proposed hardware support for TLS to run memory-monitoring functions in parallel with the original program [36]. In contrast to this prior work, Aftersight requires no hardware support and works on today’s commodity processors. Without support for TLS, current processors cannot quickly fork a new thread. Instead, Aftersight uses virtual-machine replay to continuously mirror the dynamic state of the original program on the analysis machines, thereby making it possible for them to analyze this state on spare processors or cores. Aftersight also includes new optimizations to accelerate the analysis machines by leveraging information generated by the original program.

A recent workshop paper briefly describes a similar approach to executing analyses in parallel with the original program [6]. As with TLS, their system requires hardware support to mirror the dynamic state of the original program onto spare processors by logging detailed data from each instruction, including the instruction counter, type, and input/output identifiers. The large volume of log data slows performance down by 4–10x. In contrast, Aftersight requires no hardware support, and runs with low overhead.

Other research has looked at combining simulators and hypervisors. For example, Ho, et al. [12] allowed switching back and forth between QEMU and Xen, and SimOS allows users to switch between direct execution and detailed simulation [22]. Aftersight differs from these systems by decoupling the analyses from the main

workload and allowing both modes to run at the same time. This takes advantage of parallelism in processor cores and eliminates the user-visible overheads of analysis.

Besides decoupled analysis, there are a variety of other ways to reduce the overhead of heavyweight analysis. For example, sampling reduces analysis overhead [7] but can miss relevant events and only works for specific analyses. Hardware solutions [29] can also reduce the perturbation to the main workload caused by analysis, but requiring custom hardware makes this approach less attractive.

9 Conclusions

Dynamic program analysis has a wide range of compelling uses. Unfortunately, powerful analyses typically add substantial overhead which perturbs the workload, so the vast majority of program execution takes place with very little checking. This means that many critical software flaws remain overlooked, when they could be detected during testing, quality assurance, and deployment. Similarly, in operational settings, the high overhead of analysis deters the use of many potentially promising techniques for intrusion detection and prevention.

We have presented Aftersight, a system that helps overcome these limitations by decoupling dynamic program analysis from execution through virtual machine replay. This allows analysis to be carried out on the replayed execution, independent of the main workload. This mechanism allows several choices along the safety/performance spectrum, such as synchronous safety, best-effort safety, and offline analysis. Synchronous safety achieves performance comparable to in-line analysis, while best-effort safety and offline analysis make it possible to apply slow, expensive analysis techniques on realistic production workloads without perturbing their performance.

We discussed how Aftersight supports the use of different record and replay platforms and the benefits of allowing each to be independently optimized based on their need for performance or extensibility. We presented our prototype of Aftersight, and evaluated it with several online and offline analyses.

Dynamic program analysis is a promising technique for solving many problems. However, without a means of overcoming its performance costs, it will continue to see limited use. In light of the ubiquitous adoption of

virtualization technology, we believe decoupled analysis, as demonstrated by Aftersight, offers a promising approach to enabling the use of this technique in a much broader set of applications.

References

[1] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In ASPLOS, pages 2 - 13, October 2006.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers Principles, Techniques, and Tools. Addison-Wesley, Reading, MA, 1986.

[3] F. Bellard. QEMU, A Fast and Portable Dynamic Translator. In Proc. USENIX Annual Technical Conference, pages 41 - 41, Berkeley, CA, USA, 2005. USENIX Association.

[4] T. C. Bressoud and F. B. Schneider. Hypervisor-based Fault Tolerance. ACM Trans. Comput. Syst., 14(1):80 - 107, 1996.

[5] F. Chang and G. A. Gibson. Automatic I/O hint generation through speculative execution. In Proceedings of the 1999 Symposium on Operating Systems Design and Implementation (OSDI), February 1999.

[6] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry,

R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and

S. W. Schlosser. Log-Based Architectures for General-Purpose Monitoring of Deployed Code. 2006 Workshop on Architectural and System Support for Improving Software Dependability, October 2006.

[7] T. M. Chilimbi and M. Hauswirth. Low-Overhead Memory Leak Detection Using Adaptive Statistical Profiling. In ASPLOS, October 2004.

- [8] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In Proc. 1998 SIGMETRICS Symposium on Parallel and distributed tools (SPDT), August 1998.
- [9] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In OSDI, pages 211 - 224, New York, NY, USA, 2002. ACM.
- [10] S. L. Graham, P. B. Kessler, and M. E. McKusick. Gprof: A Call Graph Execution Profiler. In SIGPLAN '82 Symposium on Compiler Construction, pages 120 - 126, June 1982.
- [11] R. Hastings and B. Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In Proc. Winter 1992 USENIX Conference, pages 125 - 138, 1992.
- [12] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical Taint-Based Protection Using Demand Emulation. In EuroSys, pages 29 - 41, New York, NY, USA, 2006. ACM Press.
- [13] Intel. IA-32 Intel Architecture Software Developer's Manual. Volumes I, II, and III, 2006.
- [14] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In SOSp, pages 91 - 104, October 2005.
- [15] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. In Proc. USENIX Annual Technical Conference, pages 1 - 15, 2005.
- [16] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In ISCA, pages 284 - 295, Washington, DC, USA, 2005. IEEE Computer Society.
- [17] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In

PLDI, pages 89 - 100, New York, NY, USA, 2007. ACM Press.

[18] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In Proc. Network and Distributed System Security Symposium (NDSS), 2005.

[19] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing Security Checks on Commodity Hardware. In ASPLOS, March 2008.

[20] J. Oplinger and M. S. Lam. Enhancing Software Reliability using Speculative Threads. In ASPLOS, pages 184 - 196, October 2002.

[21] H. Patil and C. N. Fischer. Efficient Run-time Monitoring Using Shadow Processing. In Proc. International Workshop on Automated and Algorithmic Debugging (AADEBUG), pages 119 - 132, May 1995.

[22] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. Modeling and Computer Simulation, 7(1):78 - 103, 1997.

[23] M. Rosenblum and T. Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. IEEE Computer, May 2005.

[24] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM Transactions on Computer Systems, 15(4):391 - 411, 1997.

[25] B. A. Schroeder. On-Line Monitoring: a Tutorial. IEEE Computer, 28(6):72 - 78, June 1995.

[26] J. Seward and N. Nethercote. Using Valgrind to Detect Undefined Value Errors With Bit-Precision. In Proc. USENIX Annual Technical Conference, pages 2 - 2, Berkeley, CA, USA, 2005. USENIX Association.

- [27] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. Flashback: A light-weight rollback and deterministic replay extension for software debugging. In Proc. USENIX Technical Conference, June 2004.
- [28] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In USENIX Annual Technical Conference, pages 1 - 14, 2001.
- [29] J. J. P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi. A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging. IEEE Transactions on Software Engineering, 16(8):897 - 916, August 1990.
- [30] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In OSDI, pages 181 - 194, December 2002.
- [31] S. Wallace and K. Hazelwood. SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance. In Proc. 2007 International Symposium on Code Generation and Optimization (CGO), pages 209 - 217, March 2007.
- [32] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In OSDI, December 2004.
- [33] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. SIGMETRICS Perform. Eval. Rev., 24(1):68 - 79, 1996.
- [34] M. Xu, R. Bodik, and M. D. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In ISCA, pages 122 - 135, New York, NY, USA, 2003. ACM Press.
- [35] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weiss-man. ReTrace: Collecting Execution Trace

with Virtual Machine Deterministic Replay. In Proc. 2007 Workshop on Modeling, Benchmarking and Simulation (MoBS), June 2007.

[36] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In ISCA, June 2004.

Notes

1First released in VMware Workstation 6, where it was an experimental feature.

2These add enough overhead that QEMU completely ignores the behavior. This turns out to work for many workloads, but not all. Playing fast and loose with the specification in this way inevitably causes failures—non-executable stacks are popularly implemented with segment limits in major x86 Unix derivatives where non-executable page protections are unavailable (which is true for all non-PAE kernels), but QEMU makes them behave incorrectly.

Bridging the Gap between Software and Hardware Techniques
for I/O Virtualization

Jose Renato Santos† Yoshio Turner† G.(John) Janakiramah 1 Ian Pratt §

†Hewlett Packard Laboratories, Palo Alto, CA

§ University of Cambridge, Cambridge, UK

Abstract

The paravirtualized I/O driver domain model, used in Xen, provides several advantages including device driver isolation in a safe execution environment, support for guest VM transparent services including live migration, and hardware independence for guests. However, these advantages currently come at the cost of high CPU overhead which can lead to low throughput for high bandwidth links such as 10 gigabit Ethernet. Direct I/O has been proposed as the solution to this performance problem but at the cost of removing the benefits of the driver domain model. In this paper we show how to significantly narrow the performance gap by improving the performance of the driver domain model. In particular, we reduce execution costs for conventional NICs by 56% on the receive path, and we achieve close to direct I/O performance for network devices supporting multiple hardware receive queues. These results make the Xen driver domain model an attractive solution for I/O virtualization for a wider range of scenarios.

1 Introduction

In virtual machine environments like VMware [12], Xen [7], and KVM [23], a major source of performance degradation is the cost of virtualizing I/O devices to allow multiple guest VMs to securely share a single device. While the techniques used for virtualizing CPU and memory resources have very low overhead leading to near native performance [29][7][4], it is challenging to efficiently virtualize most current I/O devices. Each interaction between a guest OS and an I/O device needs to undergo costly interception and validation by the virtualization layer for security isolation and for data multiplexing and demultiplexing [28]. This problem is particularly acute when virtualizing high-bandwidth network interface devices because frequent software interactions with the device

are needed to handle the high rate of packet arrivals.

Paravirtualization [30] has been proposed and used (e.g., in Xen [7][13]) to significantly shrink the cost and complexity of I/O device virtualization compared to using full device emulation. In this approach, the guest OS executes a paravirtualized (PV) driver that operates on a simplified abstract device model exported to the guest. The real device driver that actually accesses the hardware can reside in the hypervisor, or in a separate device driver domain which has privileged access to the device hardware. Using device driver domains is attractive because they allow the use of legacy OS device drivers for portability, and because they provide a safe execution environment isolated from the hypervisor and other guest VMs [16][13]. Even with the use of PV drivers, there remains very high CPU overhead (e.g., factor of four) compared to running in non-virtualized environments [18] [17], leading to throughput degradation for high bandwidth links (e.g., 10 gigabits/second Ethernet). While there has been significant recent progress making the transmit path more efficient for paravirtualized I/O [17], little has been done to streamline the receive path, the focus of this paper.

To avoid the high performance overheads of software-based I/O device virtualization, efforts in academia and industry are working on adding, to varying degrees, hardware support for virtualization into I/O devices and platforms [10][24][32][22][3][6]. These approaches present a tradeoff between efficiency and transparency of I/O device virtualization. In particular, using hardware support for “direct I/O” [32][24][22], in which a device presents multiple logical interfaces which can be securely accessed by guest VMs bypassing the virtualization layer, results in the best possible performance, with CPU cost close to native performance. However, direct I/O lacks key advantages of a dedicated driver domain model: device driver isolation in a safe execution environment avoiding guest domain corruption by buggy drivers, etc., and full support for guest VM transparent services including live migration [27] [21] [11] and traffic monitoring. Restoring these services would require either additional support in the devices or breaking the transparency of the services to the guest VM. In addition, it is difficult to exploit direct I/O in emerging virtual appliance models of software distribution which rely on the ability to execute on arbitrary hardware platforms. To use direct I/O, virtual appliances would have to include device drivers for a large variety of devices increasing their complexity, size, and maintainability.

In this paper, we significantly bridge the performance gap between the driver domain model and direct I/O in the

Xen virtual machine environment, making the driver domain model a competitive and attractive approach in a wider range of scenarios. We first present a detailed analysis of the CPU costs of I/O operations for devices without hardware support for virtualization. Based on this analysis we present implementation and configuration optimizations, and we propose changes to the software architecture to reduce the remaining large costs identified by the analysis: per-packet overheads for memory management and protection, and per-byte data copy overheads. Our experimental results show that the proposed modifications reduce the CPU cost by 56% for a streaming receive microbenchmark. In addition to improving the virtualization of conventional network devices, we propose extensions to the Xen driver domain model to take advantage of emerging network interface devices that provide multiple hardware receive queues with packet demultiplexing performed by the device based on packet MAC address and VLAN ID [10]. The combination of multi-queue devices with our software architecture extensions provides a solution that retains all the advantages of the driver domain model and preserves all the benefits of virtualization including guest-transparent migration and other services. Our results show that this approach has low overhead, incurring CPU cost close to that of direct I/O for a streaming receive microbenchmark.

The rest of the paper is organized as follows. Section 2 reviews the Xen driver domain model. Section 3 presents a detailed analysis and breakdown of the costs of I/O virtualization. Section 4 presents our implementation and architectural changes that significantly improve performance for the driver domain model. Section 5 discusses how configuration settings affect I/O virtualization performance, and Section 6 presents our conclusions.

2 Xen Network I/O Architecture

Figure 1 shows the architecture of Xen paravirtualized (PV) networking. Guest domains (i.e., running virtual machines) host a paravirtualized device driver, netfront, which interacts with the device indirectly through a separate device driver domain, which has privileged access to the hardware. Driver domains directly access hardware devices that they own; however, interrupts from these devices are first handled by the hypervisor which then notifies the corresponding driver domain through virtual interrupts. Netfront communicates with a counterpart backend driver called netback in the driver domain, using shared memory I/O channels. The driver domain uses a software bridge to route packets among the physical device and multiple guests through their netback interfaces.

Each I/O channel comprises of an event notification mechanism and a bidirection ring of asynchronous requests carrying I/O buffer descriptors from netfront to netback and the corresponding responses. The event notification mechanism enables netfront and netback to trigger a virtual interrupt on the other domain to indicate new requests or responses have been posted. To enable driver domains to access I/O buffers in guest memory Xen provides a page grant mechanism. A guest creates a grant reference providing access to a page and forwards the reference as part of the I/O buffer descriptor. By invoking a hypercall, the driver domain uses the grant reference to access the guest page. For transmit (TX) requests the driver domain uses a hypercall to map the guest page into its address space before sending the request through the bridge. When the physical device driver frees the page a callback function is automatically invoked to return a response to netfront which then revokes the grant. For RX requests netfront posts I/O buffer page grants to the RX I/O channel. When netback receives a packet from the bridge it retrieves a posted grant from the I/O channel and issues a grant copy hypercall to copy the packet to the guest page. Finally, netback sends a response to the guest via the RX channel indicating a packet is available.

Figure 1: Xen PV driver architecture

3 Xen Networking Performance Analysis

Table 1: Classes grouping Linux functions

| Class | Description |
|-----------|------------------------------------|
| driver | network device driver and netfront |
| network | general network functions |
| bridge | network bridge |
| netfilter | network filter |
| netback | netback |
| mem | memory management |
| interrupt | interrupt, softirq, & Xen events |
| schedule | process scheduling & idle loop |

syscall system call
time time functions
dma dma interface
hypercall call into Xen
grant issuing & revoking grant

Table 2: Classes grouping Xen Function

| Class | Description |
|-----------|---|
| grant | grant map unmap or copy operation |
| schedule | domain scheduling |
| hypercall | hypercall handling |
| time | time functions |
| event | Xen events |
| mem | memory |
| interrupt | interrupt |
| entry | enter/exit Xen (hypercall, interrupt, fault), |
| traps | fault handling (also system call intercept) |

Table 3: Global function grouping

| Class | Description |
|-----------|-----------------------------------|
| xen0 | Xen functions in domain 0 context |
| kernel0 | kernel functions in domain 0 |
| grantcopy | data copy in grant code |
| xen | Xen functions in guest context |
| kernel | kernel functions in guest |
| usercopy | copy from kernel to user buffer |

This section presents a detailed performance analysis of Xen network I/O virtualization. Our analysis focuses on the receive path, which has higher virtualization overhead than the transmit path and has received less attention in the literature. We quantify the cost of processing network packets in Xen and the distribution of cost among the various components of the system software. Our analysis compares the Xen driver domain model, the Xen direct I/O model, and native Linux. The analysis provides insight into the main sources of I/O virtualization overhead and guides the design changes and optimizations we present in Section 4.

3.1 Experimental Setup

We run our experiments on two HP c-class blade servers BL460c connected through a gigabit Cisco Catalyst Blade Switch 3020. Each server has two 3GHz Intel Xeon 5160 CPUs (two dual-core CPUs) with 4MB of L2 cache each, 8GB of memory, and two Broadcom NetXtreme II BCM57085 gigabit Ethernet Network Interface Cards (NICs). Although each server had two NICs, only one was used in each experiment presented in this paper.

To generate network traffic we used the netperf[1] UDP_STREAM microbenchmark. It would be difficult to separate the CPU costs on the transmit and receive paths using TCP, which generates ACK packets in the opposite direction of data packets. Therefore, we used unidirectional UDP instead of TCP traffic. Although all results are based on UDP traffic, we expect TCP to have similar behavior at the I/O virtualization level.

We used a recent Xen unstable2 distribution with paravirtualized Linux domains (i.e. a modified Linux kernel that does not require CPU virtualization support) using linux-2.6.18-xen3. The system was configured with one guest domain and one privileged domain 0 which was also used as driver domain. For direct I/O evaluation the application was executed directly in domain 0. Both domain 0 and the guest domain were configured with 512MB of memory and a single virtual CPU each. The virtual CPUs of the guest and domain 0 were pinned to different cores of different CPU sockets⁴.

We use OProfile [2][18] to determine the number of CPU cycles used in each Linux and Xen function when processing network packets. Given the large number of kernel and hypervisor functions we group them into a

small number of classes based on their high level purpose as described in Tables 1 and 2. In addition, when presenting overall results we group the functions in global classes as described in Table 3.

To provide safe direct I/O access to guests an IOMMU [8] is required to prevent device DMA operations from accessing other guests' memory. However, we were not able to obtain a server with IOMMU support. Thus, our results for direct I/O are optimistic since they do not include IOMMU overheads. Evaluations of IOMMU overheads are provided in [9][31].

3.2 Overall Cost of I/O Virtualization

Figure 2 compares the CPU cycles consumed for processing received UDP packets in Linux, and in Xen with direct I/O access from the guest and with Xen paravirtualized (PV) driver.

The graph shows results for three different sizes of UDP messages: 52, 1500 and 48000 bytes. A message with 52 bytes corresponds to a typical TCP ACK, while a message with 1500 bytes corresponds to the maximum ethernet packet size. Using messages with 48000 bytes also generates maximum size packets but reduces the overhead of system calls at the kernel and application interface by delivering data in larger chunks. The experiments with maximum size packets in Figure 2 and in the remainder results presented in this paper were able to saturate the gigabit link. Experiments with 52-byte packets were not able to saturate the network link as the receive CPU becomes the bottleneck. To avoid having an overloaded CPU with a high number of dropped packets, we throttled the sender rate for small packets to have the same packet rate as with the large packet sizes.

The results show that in the current Xen implementation, the PV driver model consumes significantly more CPU cycles to process received packets than Linux. Direct I/O has much better performance than the PV driver, but it still has significant overhead when compared to Linux, especially for small message sizes. In the rest of this paper we analyze the sources of I/O virtualization overheads in detail, and based on this analysis we propose several changes in the design and implementation of network I/O virtualization in Xen.

Figure 2: CPU usage to receive UDP packets

Figure 3: Kernel CPU cost for direct I/O

We start by looking at the overheads when running guests with direct I/O access. It is surprising that for small message sizes Xen with direct I/O uses twice the number of CPU cycles to process received packets compared to non-virtualized Linux. This is a consequence of memory protection limitations of the 64-bit x86 architecture that are not present in the 32-bit X86 architecture. The 64-bit x86 architecture does not support memory segmentation, which is used for protecting Xen memory in the 32-bit architecture. To overcome this limitation Xen uses different page tables for kernel and user level memory and needs to intercept every system call to switch between the two page tables. In our results, the overhead of intercepting system calls is negligible when using large message sizes (48000 bytes), since each system call consumes data from many received packets (32 packets with 1500 bytes). For more details on system call overheads the reader is referred to an extended version of this paper[26]. System call interception is an artifact of current hardware limitations which will be eliminated over time as CPU hardware support for virtualization [20][5] improves. Therefore, we ignore its effect in the rest of this paper and discuss only results for large message sizes (48000 bytes).

3.3 Analysis of Direct I/O Performance

Ignoring system call interception we observe that the kernel CPU cost for Xen with direct I/O is similar to Linux, as illustrated in the large message results in Figure 3. Xen with direct I/O consumes approximately 900 more CPU cycles per packet than native Linux (i.e. 31% overhead) for the receive path.

Of these 900 cycles, 250 cycles are due to paravirtualization changes in the kernel code. In particular, direct I/O in Xen has more CPU cycles compared to Linux in DMA functions. This is due to the use of a different implementation of the Linux DMA interface. The DMA interface is a kernel service used by device drivers to translate a virtual address to a bus address used in device DMA operations. Native Linux uses the default (`pci-nommu.c`) implementation, which simply returns the physical memory address associated with the virtual address. Para-virtualized Linux uses the software emulated I/O TLB code (`swiotlb.c`), which implements the Linux bounce

buffer mechanism. This is needed in Xen because guest I/O buffers spanning multiple pages may not be contiguous in physical memory. The I/O bounce buffer mechanism uses an intermediary contiguous buffer and copies the data to/from its original memory location after/before the DMA operation, in case the original buffer is not contiguous in physical memory. However, the I/O buffers used for regular ethernet packets do not span across multiple pages and thus are not copied into a bounce buffer. The different number of observed CPU cycles is due to the different logic and extra checks needed to verify if the buffer is contiguous, and not due to an extra data copy.

Of the total 900 cycles/packet overhead for direct I/O, the hypervisor accounts for 650 cycles/packet as shown in Figure 6. Most of these cycles are in timer related functions, interrupt processing, and entering and exiting (entry) the hypervisor due to interrupts and hypercalls.

3.4 Analysis of PV Driver Performance

Xen PV driver consumes significantly more CPU cycles than direct I/O as shown in Figure 2. This is expected since Xen PV driver runs an additional domain to host the device driver. Extra CPU cycles are consumed both by the driver domain kernel and by the hypervisor which is now executed in the context of two domains compared to one domain with direct I/O. Additional CPU cycles are consumed to copy I/O data across domains using the Xen grant mechanism. The end result is that the CPU cost to process received network packets for Xen PV driver is approximately 18,200 cycles per packet which corresponds to 4.5 times the cost of native Linux. However, as we show in this paper, implementation and design optimizations can significantly reduce this CPU cost.

Of the total 18,200 CPU cycles consumed, approximately 1700 cycles are for copying data from kernel to user buffer (usercopy), 4300 for guest kernel functions (kernel), 900 for Xen functions in guest context (xen), 3000 for copying data from driver domain to guest domain (grantcopy), 5400 for driver domain kernel functions (kernel0), and 2900 for Xen functions in driver domain context (xen0). In contrast native Linux consumes approximately 4000 CPU cycles for each packets where 1100 cycles are used to copy data from kernel to user space and 2900 cycles are consumed in other kernel functions. In the following subsections we examine each component of the

CPU cost for Xen PV driver to identify the specific causes of high overhead.

3.4.1 Copy overhead

We note in Figure 2 that both data copies (usercopy and grantcopy) in Xen PV driver consume a significantly higher number of CPU cycles than the single data copy in native Linux (usercopy). This is a consequence of using different memory address alignments for source and destination buffers.

Intel processor manuals [15] indicate that the processor is more efficient when copying data between memory locations that have the same 64-bit word alignment and even more efficient when they have the same cache line alignment. But packets received from the network are non aligned in order to align IP headers following the typical 14-byte Ethernet header. Netback copies the non aligned packets to the beginning of the granted page which by definition is aligned. In addition, since now the packet starts at a 64-bit word boundary in the guest, the packet payload will start at a non word boundary in the destination buffer, due to the Ethernet header. This causes the second copy from the kernel to the user buffer in the guest to also be misaligned.

The two unaligned data copies consume significantly more CPU cycles than aligned copies. To evaluate this overhead, we modified netback to copy the packet into the guest buffer with an offset that makes the destination of the copy have the same alignment as the source. This is possible because we use a Linux guest which permits changing the socket buffer boundaries after it is allocated. Figure 4 shows the CPU cost of the grant copy for different copy alignments. The first bar shows the number of CPU cycles used to copy a 1500 byte packet when the source and destination have different word alignments. The second bar shows the number of CPU cycles when source and destination have the same 64-bit word alignment but have different cache line alignment (128 bytes), while the third bar shows the number of CPU cycles when source and destination have the same cache line alignment. These results show that proper alignment can reduce the cost of the copy by a factor of two.

Figure 4: Alignment effect on data copy cost

Figure 5: Kernel CPU cost

Figure 6: Hypervisor CPU cost

Figure 7: Driver domain CPU cost (kernel)

3.4.2 Kernel overhead

Figure 5 compares the kernel cost for the Xen PV driver model and for the direct I/O model. Xen PV driver uses significantly more CPU cycles than direct I/O, especially in device driver functions. While the device drivers are different (i.e. physical device driver for direct I/O and netfront for PV driver), we would expect that the physical device driver in direct I/O would be more expensive as it has to interact with a real device, while netfront in the PV driver model only accesses the I/O channels hosted in main memory. After careful investigation we determined that the main source of increased CPU cycles in Xen PV driver is the use of fragments in guest socket buffers.

Xen supports TCP Segmentation Offloading (TSO)[17] and Large Receive Offloading (LRO)[14][19] in its virtual interfaces enabling the use of large packets spanning multiple pages for efficient packet processing. For this reason netfront posts full page buffers which are used as fragments in socket buffers, even for normal size ethernet frames. Netfront copies the first 200 bytes of the first fragment into the main linear socket buffer area, since the network stack requires the packet headers in this area. Any remaining bytes past the 200 bytes are kept in the fragment. Use of fragments thereby introduces copy overheads and socket buffer memory allocation overheads.

To measure the cost of using fragments we modified netfront to avoid using fragments. This was accomplished by pre-allocating socket buffers and posting those buffers directly into the I/O channel instead of posting fragment pages. This modification assumes packets will not use multiple pages and thus will not require fragments, since the posted buffers are regular socket buffers and not full page fragments. Since our NIC does not support LRO [14] and it is not configured with jumbo packets, all packets received from the external network use single-page buffers in our experiments. Of course this modification cannot be used for guest to guest communication since a

guest can always send large fragmented packets. The third bar in Figure 5 shows the performance results using the modified netfront. The result shows that using socket buffers with fragments is responsible for most of the additional kernel cycles for the PV driver case when compared to direct I/O.

Without the overhead of fragments the kernel CPU cost for PV driver is very close to that of direct I/O, except for small variations in the distribution of cycles among the different kernel functions. In particular, netfront in Xen PV driver now has lower cost than the physical device driver in direct I/O, as expected. Also since netfront does not access a physical device it does not need to use the software I/O TLB implementation of the DMA interface used by direct I/O. On the other hand PV drivers have the cost of issuing and revoking grants, which are not used with direct I/O.

Most of this grant cost is due to the use of expensive atomic compare and swap instructions to revoke grant privileges in the guest grant table. This is necessary because Xen uses different bits of the same grant table field to store the status of the grant (grant in use or not by the driver domain) updated by Xen, and the grant access permission (enable or revoke grant access) updated by the issuing domain. The guest must revoke and check that the grant is no longer in use by the driver domain using an atomic operation, to ensure that a driver domain does not race with grant revoking and keep an undesired reference to the page after the grant is revoked. The use of atomic compare and swap instructions to revoke a grant adds a significant number of CPU cycles in the guest kernel cost. If however these two different grant bits are stored in different words, we can ensure atomicity using less expensive operations such as memory barriers. The result with implementation optimizations presented later in Section 4 include this and other grant optimizations discussed in the following Section 3.4.3.

3.4.3 Hypervisor overhead

I/O processing consumes CPU cycles executing hypervisor functions in addition to guest kernel code. Figure 6 shows the CPU cost in Xen hypervisor functions for receiving network packets with PV driver. The graph shows the CPU cost when executing Xen functions in the context of the guest and in the context of the driver domain (i.e. domain 0), and compares them with the CPU cost for direct I/O.

The graph shows that most of the CPU cost for the hypervisor is due to code executing in the context of the driver domain, and the larger cost components are due to grant operations and schedule functions. The hypervisor CPU cost in guest context for PV drivers is similar to the hypervisor cost for direct I/O except for a higher cost in schedule functions.

The higher cost in scheduling functions for PV driver is due to increased cache misses when accessing Xen data structures for domain scheduling purposes. With PV driver, the guest and the driver domain run on different CPUs, causing domain data structures used by scheduling functions to bounce between the two CPUs.

We identified the most expensive operations for executing grant code by selectively removing code from the grant functions. Basically they are the following: 1) acquiring and releasing spin locks, 2) pinning pages, and 3) use of expensive atomic swap operations to update grant status as previously described.

All of these operations can be optimized. The number of spinlock operations can be significantly reduced by combining the operations of multiple grants in a single critical section. The use of atomic swap operations can be avoided by separating grant fields in different words as described in section 3.4.2. Note that this optimization reduces overhead in Xen and in the guest, since both of them have to access the same grant field atomically.

The cost of pinning pages can also be optimized when using grants for data copy. During a grant copy operation, the hypervisor creates temporary mappings into hypervisor address space for both source and destination of the copy. The hypervisor also pins (i.e. increment a reference counter) both pages to prevent the pages from being freed while the grant is active. However, usually one of the pages is already pinned and mapped in the address space of the current domain which issued the grant operation hypercall. Thus we can avoid mapping and pinning the domain local page and just pin the foreign page referred by the grant. It turns out that pinning pages for writing in Xen is significantly more expensive than pinning pages for read, as it requires to increment an additional reference counter using an expensive atomic instruction. Therefore, this optimization has higher performance impact when the grant is used to copy data from a granted page to a local page (as we propose below in Section 4.1) instead of the other way around.

3.4.4 Driver domain overhead

Figure 7 shows CPU cycles consumed by the driver domain kernel (2nd bar graph; domain 0) and compares it with the kernel cost in native Linux (1st bar graph). The results show that the kernel cost in the driver domain is almost twice the cost of the Linux kernel. This is somewhat surprising, since in the driver domain the packet is only processed by the lower level of the network stack (ethernet), although it is handled by two device drivers: native device driver and netback. We observe that a large number of the CPU cycles in driver domain kernel are due to bridge and netfilter functions. Note that although the kernel has netfilter support enabled, no netfilter rule or filter is used in our experiments. The cost shown in the graph is the cost of netfilter hooks in the bridge code that are always executed to test if a filter needs to be applied. The third bar graph in the figure shows the performance of a driver domain when the kernel is compiled with the bridge netfilter disabled. The results show that most of the bridge cost and all netfilter cost can be eliminated if the kernel is configured appropriately when netfilter rules are not needed.

4 Xen Network Design Changes

In the previous section we identified several sources of overhead for Xen PV network drivers. In this section we propose architectural changes to the Xen PV driver model that significantly improve performance. These changes modify the behavior of netfront and netback, and the I/O channel protocol.

Some inefficiencies identified in Section 3 can be reduced through implementation optimizations that do not constitute architectural or protocol changes. Although some of the implementation optimizations are easier to implement in the new architecture, we evaluated their performance impact in the current architecture in order to separate their performance benefits from that of the architectural changes. The implementation optimizations include: disabling the netfilter bridge, using aligned data copies, avoiding socket buffer fragments and the various grant optimizations discussed in Sections 3.4.2 and 3.4.3.

The second bar in Figure 8 shows the cumulative performance impact of all these implementation optimizations and is used as a reference point for the performance improvements of the architectural changes presented in this

section. In summary, the implementation optimizations reduce the CPU cost of Xen PV driver by approximately 4950 CPU cycles per packet. Of these, 1550 cycles are due to disabling bridge netfilter, 1850 cycles due to using cache aligned data copies, 900 cycles due to avoiding socket buffer fragments and 650 cycles due to grant optimizations.

4.1 Move Data Copy to Guest

As described in Section 3 a primary source of overhead for Xen PV driver is the additional data copy between driver domain and guest. In native Linux there is only one data copy, as the received packet is placed directly into a kernel socket buffer by the NIC and later copied from there to the application buffer. In Xen PV driver model there are two data copies, as the received packet is first placed in kernel memory of the driver domain by the NIC, and then it is copied to kernel memory of the guest before it can be delivered to the application.

This extra cost could be avoided if we could transfer the ownership of the page containing the received packet from the driver domain to the guest, instead of copying the data. In fact this was the original approach used in previous versions of Xen [13]. The first versions of Xen PV network driver used a page flipping mechanism which swapped the page containing the received packet with a free guest page, avoiding the data copy. The original page flip mechanism was replaced by the data copy mechanism in later versions of Xen for performance reasons. The cost of mapping and unmapping pages in both guest and driver domain was equivalent to the copy cost for large 1500 byte packets, which means that page flipping was less efficient than copy for small packet sizes [25]. In addition, the page flip mechanism increases memory fragmentation and prevents the use of super-page mappings with Xen. Menon et al. [17] have shown that super-pages provide superior performance in Xen, making page flipping unattractive.

One problem with the current data copy mechanism is that the two data copies per packet are usually performed by different CPUs leading to poor data cache behavior. On an SMP machine, it is expected that an I/O intensive guest and the driver domain will be executing on different CPUs, especially if there is high I/O demand. The overhead introduced by the extra data copy can be reduced if both copies are performed by the same CPU and benefit from cache locality. The CPU will bring the packet to its cache during the first data copy. If the data is

still present in the CPU cache during the second copy, this copy will be significantly faster using fewer CPU cycles. Of course there is no guarantee that the data will not be evicted from the cache before the second copy, which can be delayed arbitrarily depending on the application and overall system workload behavior. At high I/O rates, however, it is expected that the data will be delivered to the application as soon as it is received and will benefit from L2 cache locality.

We modified the architecture of Xen PV driver and moved the grant copy operation from the driver domain to the guest domain, improving cache locality for the second data copy. In the new design, when a packet is received netback issues a grant for the packet page to the guest and notifies the guest of the packet arrival through the I/O channel. When netfront receives the I/O channel notification, it issues a grant copy operation to copy the packet from a driver domain page to a local socket buffer, and then delivers the packet to the kernel network stack.

Moving the grant copy to the guest has benefits beyond speeding up the second data copy. It avoids polluting the cache of the driver domain CPU with data that will not be used, and thus should improve cache behavior in the driver domain as well. It also provides better CPU usage accounting. The CPU cycles used to copy the packet will now be accounted to the guest instead of to the driver domain, increasing fairness when accounting for CPU usage in Xen scheduling. Another benefit is improved scalability for multiple guests. For high speed networks (e.g. 10GigE), the driver domain can become the bottleneck and reduce I/O throughput. Offloading some of the CPU cycles to multiple guests' CPUs avoids the driver domain from becoming a bottleneck improving I/O scalability. Finally, some implementation optimizations described earlier are easier to implement when the copy is done by the guest. For example, it is easier to avoid the extra socket buffer fragment discussed in Section 3.4.2. Moving the copy to the guest allows the guest to allocate the buffer after the packet is received from netback at netfront. Having knowledge of the received packet allows netfront to allocate the appropriate buffers with the right sizes and alignments. Netfront can allocate one socket buffer for the first page of each packet and additional fragment pages only if the packet spans multiple pages. For the same reason, it is also easier to make aligned data copies, as the right socket buffer alignment can be selected at buffer allocation time.

The third bar in Figure 8 shows the performance benefit of moving the grant copy from the driver domain to the guest. The cost of the data copy is reduced because of better use of the L2 cache. In addition, the number of cycles consumed by Xen in guest context (xen) increases while it decreases for Xen in driver domain context (xen0). This is because the cycles used by the grant operation are shifted from the driver domain to the guest. We observe that the cost decrease in xen0 is higher than the cost increase in xen leading to an overall reduction in the CPU cost of Xen. This is because the grant optimizations described in Section 3.4.3 are more effective when the grant operations are performed by the guest, as previously discussed.

In summary, moving the copy from the driver domain to the guest reduces the CPU cost for Xen PV driver by approximately 2250 cycles per packet. Of these, 1400 cycles are due to better cache locality for guest copies (usercopy, grantcopy, and also kernel for faster accesses to packet headers in protocol processing), 600 cycles are due to grant optimizations being more effective (xen + xen0) and 250 cycles are due to less cache pollution in driver domain (kernel0).

4.2 Extending Grant Mechanism

Moving the grant copy to the guest requires a couple of extensions to the Xen grant mechanism. We have not yet implemented these extensions but we discuss them here.

To support copy in the receiving guest, the driver domain has to issue a grant to the memory page containing the received packet. This works fine for packets received on the physical device since they are placed in driver domain memory. In contrast, the memory buffers with packets received from other guests are not owned by the driver domain and cannot be granted to the receiving guest using the current Xen grant mechanism. Thus, this mechanism needs to be extended to provide grant transitivity allowing a domain that was granted access to another domain's page, to transfer this right to a third domain.

We must also ensure memory isolation and prevent guests from accessing packets destined to other guests. The main problem is that the same I/O buffer can be reused to receive new packets destined to different guests. When a small sized packet is received on a buffer previously used to receive a large packet for another domain,

the buffer may still contain data from the old packet. Scrubbing the I/O pages after or before every I/O to remove old data would have a high overhead wiping out the benefits of moving the copy to the guest. Instead we can just extend the Xen grant copy mechanism with offset and size fields to constrain the receiving guest to access only the region of the granted page containing the received packet data.

Figure 8: Xen PV driver Optimizations

4.3 Support for Multi-queue Devices

Although moving the data copy to the guest reduces the CPU cost, eliminating the extra copy altogether should provide even better performance. The extra copy can be avoided if the NIC can place received packets directly into the guest kernel buffer. This is only possible if the NIC can identify the destination guest for each packet and select a buffer from the respective guest's memory to place the packet. As discussed in Section 1, NICs are now becoming available that have multiple receive (RX) queues that can be used to directly place received packets in guest memory [10]. These NICs can demultiplex incoming traffic into the multiple RX queues based on the packet destination MAC address and VLAN IDs. Each individual RX queue can be dedicated to a particular guest and programmed with the guest MAC address. If the buffer descriptors posted at each RX queue point to kernel buffers of the respective guest, the device can place incoming packets directly into guest buffers, avoiding the extra data copy.

Figure 9: Multi-queue device support

Figure 9 illustrates how the Xen PV network driver model can be modified to support multi-queue devices. Netfront posts grants to I/O buffers for use by the multi-queue device drivers using the I/O channel. For multi-queue devices the driver domain must validate if the page belongs to the (untrusted) guest and needs to pin the page for the I/O duration to prevent the page being reassigned to the hypervisor or other guests. The grant map and unmap operations accomplish these tasks in addition to mapping the page in the driver domain. Mapping the page is needed for guest to guest traffic which traverses the driver domain network stack (bridge). Experimental results not presented here due to space limitations show that the additional cost of mapping the page is small

compared to the overall cost of the grant operation.

Netfront allocates two different types of I/O buffers which are posted to the I/O channel: regular socket buffers with the right alignments required by the network stack, and full pages for use as fragments in non linear socket buffers. Posting fragment pages is optional and just needed if the device can receive large packets spanning multiple pages, either because it is configured with jumbo frames or because it supports Large Receive Offload (LRO). Netback uses these grants to map the guest pages into driver domain address space buffers and keeps them in two different pools for each guest: one pool for each type of page. These buffers are provided to the physical device driver on demand when the driver needs to post RX descriptors to the RX queue associated with the guest. This requires that the device driver use new kernel functions to request I/O buffers from the guest memory. This can be accomplished by providing two new I/O buffer allocation functions in the driver domain kernel, `vmq_alloc_skb()` and `vmq_alloc_page()`. These are equivalent to the traditional Linux functions `netdev_alloc_skb()` and `alloc_page()` except that they take an additional parameter specifying the RX queue for which the buffer is being allocated. These functions return a buffer from the respective pool of guest I/O buffers. When a packet is received the NIC consumes one (or more in case of LRO or jumbo frame) of the posted RX buffers and places the data directly into the corresponding guest memory. The device driver is notified by the NIC that a packet arrived and then forwards the packet directly to netback. Since the NIC already demultiplexes packets, there is no need to use the bridge in the driver domain. The device driver can instead send the packet directly to netback using a new kernel function `vmq_netif_rx()`. This function is equivalent to the traditional Linux `netif_rx()` typically used by network drivers to deliver received messages, except that the new function takes an additional parameter that specifies which RX queue received the packet.

Note that these new kernel functions are not specific to Xen but enable use of multi-queue devices with any virtualization technology based on the Linux kernel, such as for example KVM[23]. These new functions only need to be used in new device drivers for modern devices that have multi-queue support. These new functions extend the current interface between Linux and network devices enabling the use of multi-queue devices for virtualization. This means that the same device driver for a multi-queue device can be used with Xen, KVM or any other Linux based virtualization technology.

We observe that the optimization that moves the copy to the guest is still useful even when using multi-queue devices. One reason is that the number of guests may exceed the number of RX queues causing some guests to share the same RX queue. In this case guests that share the same queue should still use the grant copy mechanism to copy packets from the driver domain memory to the guest. Also, grant copy is still needed to deliver local guest to guest traffic. When a guest sends a packet to another local guest the data needs to be copied from one guest memory to another, instead of being sent to the physical device. In these cases moving the copy to the guest still provides performance benefits. To support receiving packets from both the multi-queue device and other guests, netfront receives both types of packets on the RX I/O channel shown in Figure 9. Packets from other guests arrive with copy grants that are used by netfront, whereas packets from the device use pre-posted buffers.

We have implemented a PV driver prototype with multi-queue support to evaluate its performance impact. However, we did not have a NIC with multi-queue support available in our prototype. We used instead a traditional single queue NIC to estimate the benefits of a multi-queue device. We basically modified the device driver to dedicate the single RX queue of the NIC to the guest. We also modified netback to forward guest buffers posted by netfront to the physical device driver, such that the single queue device could accurately emulate the behavior of a multi-queue device.

The fourth bar in Figure 8 shows the performance impact of using multi-queue devices. As expected the cost of grant copy is eliminated as now the packet is placed directly in guest kernel memory. Also the number of CPU cycles in xen for guest context is reduced since there is no grant copy operation being performed. On the other hand the driver domain has to use two grant operations per packet to map and unmap the guest page as opposed to one operation for the grant copy, increasing the number of cycles in xen0 and reducing the benefits of removing the copy cost. However, the number of CPU cycles consumed in driver domain kernel is also reduced when using multi-queue devices. This is due to two reasons. First, packets are forwarded directly from the device driver to the guest avoiding the forwarding costs in the bridge. Second, since netback is now involved in both allocating and deallocating socket buffer structures for the driver domain, it can avoid the costs of their allocation

and deallocation. Instead, netback recycles the same set of socket buffer structures in multiple I/O operations. It only has to change their memory mappings for every new I/O, using the grant mechanism to make the socket buffers point to the right physical pages containing the guest I/O buffers. Surprisingly, the simplifications in driver domain have a higher impact on the CPU cost than the elimination of the extra data copy.

In summary, direct placement of data in guest memory reduces PV driver cost by 700 CPU cycles per packet, and simplified socket buffer allocation and simpler packet routing in driver domain reduces the cost by additional 1150 cycles per packet. On the other hand the higher cost of grant mapping over the grant copy, increases the cost by 650 cycles per packet providing a net cost reduction of 1200 CPU cycles per packet. However, the benefit of multi-queue devices can actually be larger when we avoid the costs associated with grants as discussed in the next section.

4.4 Caching and Reusing Grants

As shown in Section 3.4.3, a large fraction of Xen CPU cycles consumed during I/O operations are due to grant operation functions. In this section we describe a grant reuse mechanism that can eliminate most of this cost.

Figure 10: Caching and reusing grants

The number of grant operations performed in the driver domain can be reduced if we relax the memory isolation property slightly and allow the driver domain to keep guest I/O buffers mapped in its address space even after the I/O is completed. If the guest recycles I/O memory and reuses previously used I/O pages for new I/O operations, the cost of mapping the guest pages using the grant mechanism is amortized over multiple I/O operations. Fortunately, most operating systems tend to recycle I/O buffers. For example, the Linux slab allocator used to allocate socket buffers keeps previously used buffers in a cache which is then used to allocate new I/O buffers. In practice, keeping I/O buffer mappings for longer times does not compromise the fault isolation properties of driver domains, as the driver domain still can only access the same set of I/O pages and no pages containing any other guest data or code.

In order to reuse grants, Xen PV driver needs to be modified as illustrated in Figure 10. Netback keeps a cache of currently mapped grants for every guest. On every RX buffer posted by the guest (when using a multi-queue device) and on every TX request, netback checks if the granted page is already mapped in its address space, mapping it only if necessary. When the I/O completes, the mapping is not removed allowing it to be reused in future I/O operations. It is important, though, to enable the guest to explicitly request that a cached grant mapping be invalidated. This may be necessary, for example if the guest repurposes the page and uses it somewhere else in the guest or if it returns the page back to the hypervisor. In that case, it is desirable to revoke the grant and unmap the granted page from the driver domain address space, in order to preserve memory isolation between driver domain and guest. A new I/O control channel between netfront and netback is used for this purpose. Netfront sends grant invalidation requests and netback sends confirmation responses after the granted page is unmapped. In summary, this mechanism preserves the isolation between driver domain and guest memory (only I/O buffer pages are shared) and avoids the overhead of mapping and unmapping pages on every I/O operation.

Since the amount of memory consumed for each grant cached in netback is relatively small when compared with the page size, the maximum number of cached grants should be limited only by kernel address space available in the driver domain. The address space reserved for the kernel is significantly larger than the size of a typical active set of I/O buffers. For example, 1GB of the Linux address space is reserved for the kernel; although some of this space is used by other kernel functions, a large fraction of this space can be used for dynamic mapping of guest I/O buffer pages. The size of the active set of I/O buffers is highly dependent on the workload, but typically it should not exceed a few megabytes. In practice, the driver domain should be able to map most of the active I/O buffer pages in its address space for a large number of guests. Thus we expect that the grant reuse mechanism will provide close to a 100% hit rate in the netback grant cache, except for unlikely scenarios with more than hundreds of guests. Thus, the overhead of grant mapping can be reduced to almost zero in practical scenarios, when the guest buffer allocation mechanism promotes buffer reuse.

We have not yet implemented the complete grant reuse mechanism described above. Instead, for evaluation purposes we implemented a simplified mechanism that avoids the use of grants at all. We modified the I/O

channel to use physical page addresses directly instead of grants to specify RX buffers. Netfront specifies the machine physical addresses of I/O buffers in the I/O channel requests, and the driver domain uses these addresses directly when programming the DMA operations. Note that this is not a safe mechanism since there is no validation that the physical page used for I/O belongs to the corresponding guest and no guarantee that the page is pinned. Thus, this mechanism is used here just for performance evaluation purposes. The mechanism completely avoids the use of grants and estimates the benefit of the grant reuse mechanism when the hit rate on cached grants is 100%. Although this is an optimistic estimation it is expected to accurately predict actual performance, since we expect to achieve close to a 100% hit rate on cached grants by using appropriate buffer allocation mechanisms. However, validation of this estimation is planned as future work.

The fifth bar in Figure 8 shows the performance benefit of caching and reusing grants in the driver domain. As expected most of the benefit comes from reduced number of cycles in xen0 used for grant operations. In summary, grant reuse reduces the cost of PV driver by 1950 CPU cycles per packet, which combined with all other optimizations reduces the cost by 10300 CPU cycles. Although the optimizations described so far provide significant cost reduction, Xen I/O virtualization still has twice the cost of native I/O in Linux. However, it is possible to reduce this cost even further by properly setting system configuration parameters as described in the next section.

5 Tuning I/O Virtualization Configuration

5.1 Decoupling Driver Domain from Domain 0

Although the current architecture of the Xen PV driver model is flexible and allows the use of dedicated driver domains used exclusively for doing I/O, in practice most Xen installations are configured with domain 0 acting as the driver domain for all physical devices. The reason is that there is still no mechanism available in Xen that leverages the fault isolation properties of dedicated driver domains. For example there is currently no available tool that automatically detects driver domain faults and restarts them. Since hosting all drivers in domain 0 is simpler to configure this is typically the chosen configuration.

However, hosting all device drivers in domain 0 prevents tuning some configuration options that optimize I/O performance. Since domain 0 is a general purpose OS it needs to support all standard Linux utilities and Xen administration tools, thus requiring a full fledged kernel with support for all features of a general purpose operating system. This limits the flexibility in configuring the driver domain with optimized I/O configuration options. For example, disabling the bridge netfilter option of the kernel significantly improves performance as shown in Section 3.4.4. However, network tools such as iptables available in standard Linux distributions do not work properly if this kernel configuration option is disabled. This has prevented standard Linux vendors such as RedHat and Novell to enable this kernel option in their standard distribution, thus preventing this I/O optimization in practice. Separating the driver domain from domain 0 allows us to properly configure the driver domain with configurations that are optimized for I/O.

Figure 11: Interrupt throttling

5.2 Reducing Interrupt Rate

The architectural changes discussed in Section 4 address two important sources of overhead in I/O virtualization: extra data copies and the Xen grant mechanism. These are the most important overheads that are directly proportional to the amount of received traffic. Data copy overheads are proportional to the number of received bytes while grant overheads are proportional to the number of received packets.

Most of the remaining overheads are proportional to the number of times the driver domain and the guest are scheduled (this is different from the number of packets since both domains can process multiple packets in a single run). These additional overheads include processing physical interrupts, virtual interrupts, event delivery, domain scheduling, hypercalls, etc. Even though some of these overheads are also present in Linux, they have a higher impact when I/O is virtualized due to the additional levels of software such as the hypervisor and the driver domain. For example, a device interrupt for a received packet causes CPU cycles to be consumed both in the hypervisor interrupt handler and in the driver domain handler. Additional CPU cycles are consumed when handling the virtual interrupt in the guest after the packet is delivered through the I/O channel. Increasing the number of packets that are processed each time the guest or the driver domain is scheduled should reduce the remaining performance overheads.

On the receive path the driver domain is typically scheduled to process I/O when an interrupt is generated by the physical device. Most network devices available today can delay the generation of interrupts until multiple packets are received and thus reduce the interrupt rate at high throughputs. The interrupt rate for received packets can be controlled by special device driver parameters usually known as interrupt coalescing parameters. Interrupt coalescing parameters specify not only the number of packets per interrupt, but also a maximum delay after the last received packet. An interrupt is generated when either the specified number of packets is received or when the maximum specified delay is reached. This mechanism allows us to limit the interrupt rate at high I/O rates while preserving low latency at low I/O rates.

Configuring device coalescing parameters enables us to change the number of packets processed in each run of the driver domain and thus amortize the overheads over a larger number of packets. All packets received in a single interrupt by the device driver are queued in netback queues before being processed. This causes netback to process packets in batches of the same size as the device driver. Since netback only notifies netfront after all packets in the batch are processed and added to the I/O channel, the device coalescing parameters also limit the virtual interrupt rate in the guest domains, and thus amortize the I/O overheads in the guest as well.

Figure 11 shows the effect of interrupt coalescing on the CPU cost of I/O virtualization. The graph shows CPU cost for four cases: 1) Optimized PV driver using a traditional network device (guest copy), 2) Optimized PV driver using a multi-queue device (multi-queue + grant reuse), 3) Direct I/O, 4) native Linux. The figure shows the CPU cost for different interrupt rates for each of the four cases. The Linux default interrupt coalescing parameters for our Broadcom NIC is 6 pkt/int (packets per interrupt) with a maximum latency of 18 μ s. While we varied the batch size from 6 to 64 pkt/int we kept the default maximum interrupt latency of 18 μ s in all results presented in Figure 11, preserving the packet latency at low throughputs.

The graph shows that the CPU cost for the PV driver is significantly reduced when the number of packets processed per interrupt is increased, while the effect is less pronounced for both Linux and Direct I/O. This result confirms that most of the remaining I/O virtualization overheads are proportional to the interrupt rate. In Linux,

the default value of 6 pkt/int performs almost as well as a large batch of 64 pkt/int. This suggests that the default interrupt coalescing parameters for network device drivers that work well for native Linux, are not the best configuration for Xen PV driver.

Experimental results not shown here due to space limitations show that interrupt coalescing achieves approximately the same CPU cost reduction for the original Xen PV driver configuration as it does for our optimized PV driver without hardware multi-queue support, i.e., approximately 4600 cycles per packet for batches of size 64. This indicates that interrupt coalescing and the other optimizations presented in this paper are complementary.

In summary, the results show that software-only optimizations reduce I/O virtualization overheads for the Xen driver domain model from 355% to 97% of the Linux cost for high throughput streaming traffic. Moreover, the use of hardware support for I/O virtualization enables us to achieve close to native performance: multi-queue devices can reduce the overhead to only 25% of the Linux cost while direct I/O has 11% overhead. The main difference is due to the cost of executing netfront and netback when using Xen PV driver. For real applications, the effective cost difference between direct I/O and the driver domain model should much be lower, since applications will use CPU cycles for additional work besides I/O processing. The low cost of the driver domain model combined with multi-queue support suggest that it is an attractive solution for I/O virtualization.

6 Conclusion

The driver domain model used in Xen has several desired properties. It isolates the address space of device drivers from guest and hypervisor code preventing buggy device drivers from causing system crashes. Also, driver domains can support guest VM transparent services such as live migration and network traffic monitoring and control (e.g. firewalls).

However, the driver domain model needs to overcome the address space isolation in order to provide I/O services to guest domains. Device drivers need special mechanisms to gain access to I/O data in other guest domains and to move the I/O data bytes to and from those domains. In Xen this is accomplished through the grant mechanism. In this paper we propose several architectural changes that reduce the performance overhead associated

with driver domains models. First we propose two mechanisms that reduce the cost of moving the I/O data bytes between guest and driver domains: 1) we increase the cache locality of the data copy by moving the copy operation to the receiving guest CPU and 2) we avoid the data copy between domains by using hardware support of modern NICs to place data directly into guest memory. Second we minimize the cost of granting the driver domain access to guest domain pages by slightly relaxing the memory isolation property to allow a set of I/O buffers to be shared between domains across multiple I/O operations. Although these architectural changes were done in the context of Xen they are applicable to the driver domain model in general.

Our work demonstrates that it is possible to achieve near direct I/O and native performance while preserving the advantages of a driver domain model for I/O virtualization. We advocate that the advantages of the driver domain model outweigh the small performance advantage of direct I/O in most practical scenarios.

In addition, this paper identified several low-level optimizations for the current Xen implementation which had surprisingly large impact on overall performance.

References

- [1] Netperf. www.netperf.org.
- [2] Oprofile. oprofile.sourceforge.net.
- [3] Abramson, D., Jackson, J., Muthrasanallur, S., Neiger, G., Regnier, G., Sankaran, R., Schoinas, I., Uhlig, R., Vembu, B., and Wiegert, J. Intel virtualization technology for directed I/O. Intel Technology Journal 10, 3 (August 2006). www.intel.com/technology/itj/2006/v10i3/.
- [4] Adams, K., and Agesen, O. A comparison of software and hardware techniques for x86 virtualization. In ASPLOS (2006), J. P. Shen and M. Martonosi, Eds., ACM, pp. 2 - 13.

- [5] Advanced Micro Devices. AMD64 architecture programmer's manual volume 2: System programming. www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf, Sept 2007.
- [6] Advanced Micro Devices, Inc. IOMMU architectural specification. www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf, Feb 2007. PID 34434 Rev 1.20.
- [7] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T. L., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In SOSP (2003), M. L. Scott and L. L. Peterson, Eds., ACM, pp. 164 - 177.
- [8] Ben-Yehuda, M., Mason, J., Krieger, O., Xenidis, J., Van Doorn, L., Mallick, A., Nakajima, J., and Wahlig, E. Utilizing IOMMUs for virtualization in Linux and Xen. In Ottawa Linux Symposium (2006).
- [9] Ben-Yehuda, M., Xenidis, J., Ostrowski, M., Rister, K., Bruemmer, A., and Van Doorn, L. The price of safety: evaluating IOMMU performance. In Ottawa Linux Symposium (2007).
- [10] Chinni, S., and Hiremane, R. Virtual machine device queues. isdlibrary.intel-dispatch.com/isd/99/VMQueues.pdf. Supported in Intel 82575 gigE and 82598 10GigE controllers.
- [11] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live migration of virtual machines. In NSDI (2005), USENIX.
- [12] Devine, S., Bugnion, E., and Rosenblum, M. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. VMware US Patent 6397242, Oct 1998.
- [13] Fraser, K., Hand, S., Neugebauer, R., Pratt, I., Warfield, A., and Williams, M. Safe hardware access with the Xen virtual machine monitor. In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS) (October 2004).

- [14] Grossman, L. Large Receive Offload implementation in Neterion 10GbE Ethernet driver. In Ottawa Linux Symposium (2005).
- [15] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual. www.intel.com/products/processor/manuals/index.htm.
- [16] LeVasseur, J., Uhlig, V., Stoess, J., and Götz, S. Unmodified device driver reuse and improved system dependability via virtual machines. In OSDI (2004), pp. 17 - 30.
- [17] Menon, A., Cox, A. L., and Zwaenepoel, W. Optimizing network virtualization in Xen. In USENIX Annual Technical Conference (June 2006).
- [18] Menon, A., Santos, J. R., Turner, Y., Janakiraman, G., and Zwaenepoel, W. Diagnosing performance overheads in the Xen virtual machine environment. In First ACM/USENIX Conference on Virtual Execution Environments (VEE'05) (June 2005).
- [19] Menon, A., and Zwaenepoel, W. Optimizing TCP receive performance. In USENIX Annual Technical Conference (June 2008).
- [20] Neiger, G., Santoni, A., Leung, F., Rodgers, D., and Uhlig, R. Intel virtualization technology: hardware support for efficient processor virtualization. Intel Technology Journal 10, 3 (August 2006). www.intel.com/technology/itj/2006/v10i3/.
- [21] Nelson, M., Lim, B.-H., and Hutchins, G. Fast transparent migration for virtual machines. In USENIX Annual Technical Conference (April 2005).
- [22] PCI SIG. I/O virtualization. www.pcisig.com/specifications/iov/.

- [23] Qumranet. KVM: Kernel-based virtualization driver. www.qumranet.com/wp/kvm_wp.pdf.
- [24] Raj, H., and Schwan, K. High performance and scalable I/O virtualization via self-virtualized devices. In HPDC (2007).
- [25] Santos, J. R., Janakiraman, G., and Turner, Y. Network optimizations for PV guests. In 3rd Xen Summit (Sept 2006).
- [26] Santos, J. R., Turner, Y., Janakiraman, G., and Pratt, I. Bridging the gap between software and hardware techniques for i/o virtualization. In HP Labs Tech Report, HPL-2008-39 (2008).
- [27] Sapuntzakis, C., Chandra, R., Pfaff, B., Chow, J., Lam, M., and Rosenblum, M. Optimizing the migration of virtual computers. In 5th Symposium on Operating Systems Design and Implementation (December 2002).
- [28] Sugerman, J., Venkitachalam, G., and Lim, B.-H. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In USENIX Annual Technical Conference, General Track (2001), Y. Park, Ed., USENIX, pp. 1 - 14.
- [29] Waldspurger, C. A. Memory resource management in VMware ESX Server. In OSDI (2002).
- [30] Whitaker, A., Shaw, M., and Gribble, S. D. Scale and performance in the Denali isolation kernel. In OSDI (2002).
- [31] Willmann, P., Cox, A. L., and Rixner, S. Protection strategies for direct access to virtualized I/O devices. In USENIX Annual Technical Conference (June 2008).
- [32] Willmann, P., Shafer, J., Carr, D., Menon, A., Rixner, S., Cox, A. L., and Zwaenepoel, W. Concurrent

direct network access for virtual machine monitors. In High Performance Computer Architecture (HPCA) (February 2007).

1

Currently at Skytap.

2

xen-unstable.hg changeset 15521:1f348e70a5af, July 10, 2007

3

linux-2.6.18-xen.hg changeset 103:a70de77dd8d3, July 10, 2007

4

Using two cores in the same CPU would have improved performance due to the shared L2 cache. We chose the worst case configuration since we cannot enforce that all guests will share the same socket with the driver domain.

5

The message size includes IP and UDP headers but does not include 14 bytes of Ethernet header per packet.

6

Even if the guest does not request that the grant be revoked Xen will not allocate the page to another guest while the grant is active, maintaining safe memory protection between guests.

This document was translated from LATEX by HEVEA.