מערכת הפעלה אסימטרית התומכת בהרצת קוד
על גבי רכיבים היקפיים מיתכנתים

חיבור לשם קבלת תואר דוקטור לפילוסופיה

מאת

ירון וינסברג

הוגש לסנט האוניברסיטה העברית בירושלים

אוקטובר 2007

עבודה זו נעשתה בהדרכתו של פרופ' דני דולב

תודות

ראשית ברצוני להודות למנחה התיזה שלי, פרופ' דני דולב, שפתח בפני את הדלת לעולם המחקר האקדמי. יכולתו של דני להתבונן ולנתח בעיות באופן יצירתי וייחודי אתגרה אותי והפכה אותי לחוקר טוב יותר.

שנית, אני מודה לדר' טל אנקר, מנחה שותף וחבר. תמיכתו של טל לאורך כל תקופת לימודי ועצותיו המקצועיות היוו מרכיב חיוני במחקר זה.

כמו כן, ברצוני להודות למספר עמיתים וחברים יקרים: דר' פיט וויקוף ומולי בן-יהודה, על הערותיהם והצעותיהם המקוריות במהלך עבודה זו. תודה מיוחדת לדר' אופיר הולדר על התעניינותו הרבה בעבודה זו ובעצותיו המועילות.

בנוסף, אני מודה לכל חברי קבוצת האלגוריתמים המבוזרים. בפרט, אני מודה לדני ביקסון, שימרית צור-דוד ודר' אריאל דליות על עזרתם ותמיכתם.

אני מודה להורי, חוה וברקו וינסברג על אהבתם ותמיכתם במשך כל שנות לימודי ובכלל.

יותר מכל, ברצוני להודות לאשתי קרן וילדי המקסימים: יובל ותמר, על אהבתם וסבלנותם במשך לימודי הדוקטורט.

תקציר

בשנים האחרונות כוח העיבוד הקיים במחשבים האישיים גדל בקצב מסחרר. הטכנולוגיה הקיימת היום מאפשרת להגדיל את מספר הטרנזיסטורים המצוי על פרוסת הסיליקון בצורה משמעותית. כוח החישוב הקיים ביחידות העיבוד המרכזיות ובמעבדים המצויים על כרטיסים היקפיים מוכפל כל שמונה עשרה חודשים ("חוק מור") ואף מהר מזה. אולם, למרות קצב זה, מעט מאוד יישומים ומערכות מחשוב מנצלות את העובדה שגם לכרטיסים ההיקפיים, המצויים בכל מחשב אישי ושרת, יש כוח חישוב משמעותי שמרביתו איננו מנוצל.

מערכות ההפעלה המובילות כיום, אינן מאפשרות הרצת קוד על גבי רכיבים היקפיים למרות שאלה ניתנים לתכנות בצורה פשוטה למדי. לדוגמא, כרטיסים גראפיים (הנקראים לעיתים מאיצים גראפיים) מסוגלים לבצע פעולות על גבי מטריצות במהירות הגבוהה מזו של המעבד המרכזי. כרטיסים אלו ניתנים לתכנות בצורה פשוטה למדי אולם כיום הם בעיקר מנוצלים ע"י מפתחי משחקים ואפליקציות גראפיות מתקדמות (על ידי ספריות סטנדרטיות כדוגמת DirectX). כרטיסים נוספים המכילים מעבדים מהירים כוללים בין היתר כרטיסי רשת (חלקם כבר מריצים פרוטוקולי תקשורת כדוגמת TCP), בקרי דיסקים וכרטיסי הצפנה.

מחקר זה מציע מודל חדשני המאפשר הרצת אפליקציה על המעבד הראשי ועל מעבדים המצויים ברכיבים ההיקפיים. הרעיון הבסיסי מאחורי מחקר זה הוא שכל מעבד המצוי במחשב, מעבד מרכזי או מעבד על כרטיס היקפי, הוא פוטנציאל להרצת קוד של אפליקציה. במודל זה, אפליקציה יכולה לנצל ייחודיות של כרטיס היקפי או מעבד מסוים על מנת לשפר את ביצועיה.

מודל זה מנוגד למודל הקלאסי על פיו פותחו ומומשו מערכות ההפעלה עד כה. במודל הקלאסי מערכת ההפעלה הנה היישות היחידה המורשת לנהל את חומרת המחשב: המעבדים וכל חומרה היקפית אחרת כגון: עכבר,מקלדת, דיסק, כרטיס מסך וכו'. מערכת ההפעלה איננה מאפשרת גישה ישירה לחומרה משיקולי אבטחה ואמינות. בעבר הלא רחוק, החומרה ההיקפית הייתה פשוטה למדי עם אפשרויות מוגבלות לשינוי והתאמה לאפליקציה מסוימת. אולם, היות ואין זה המצב היום, יש צורך מהותי בעדכון המודל והתפיסה השמרנית על מנת לתמוך ברכיבים ההיקפיים שלעיתים אף חזקים יותר מהמעבדים הראשיים.

מחקר זה עוסק באפיון מערכת הפעלה אסימטרית המאפשרת הרצת קוד על גבי רכיבים היקפיים מיתכנתים באופן דינאמי. התמיכה הנדרשת היא הן ברמת מערכת ההפעלה (במחשב וברכיבים ההיקפיים) והן ברמת כלי הפיתוח ומודל התכנות. מודל התכנות חייב לאפשר למפתח לנצל את המשאבים העומדים לרשותו בצורה אופטימאלית ואינטואיטיבית.

הרצת קוד על גבי רכיבים היקפיים טומנת בחובה יתרונות רבים:

1. <u>רוחב פס למערכת הזיכרון</u> - למרות שכוח החישוב גדל בצורה משמעותית, המנשק למערכת הזיכרון במחשב נותר צוואר הבקבוק. אפליקציות רבות כדוגמת "חומת-אש" ואנטי-וירוס, מבצעות פעולות רבות מול מערכת הזיכרון עבור מידע כלשהו ולעיתים תכופות לא נעשה כלל שימוש במידע זה. פעולות המתבצעות ברכיבים ההיקפיים תאפשרנה סינון חבילות ומידע עוד לפני שהגיעו ליחידת הזיכרון הראשית.
2. <u>זמן אמת</u> – פעולות המבוצעות על גבי חומרה היקפית תאפשרנה תיזמונים המתקרבים למערכות זמן אמת היות ומערכות ההפעלה המורצות שם הן בדרך כלל מערכות הפעלה התומכות בזמן אמת.
3. <u>צריכת הספק</u> – מאמץ גדול מופנה היום בתעשייה להקטנת צריכת ההספק של מערכות מיחשוב. חלק מהמעבדים המיוצרים כיום תומכים במצבי חיסכון באנרגיה. כרטיסים היקפיים חסכוניים בצורה משמעותית מהמעבדים המרכזיים (לדוגמא, מעבד אינטל 4, 2.8 GHZ, צורך 68 ווט בעוד שמעבד אינטל XScale, הנפוץ ברכיבים היקפיים, צורך 0.5 ווט) ולכן הורדת פעולות אליהם תקטין את צריכת ההספק הכוללת.
4. <u>אבטחה</u> – הפעלת קוד על גבי רכיבים היקפיים תקשה ואולי אף תמנע התקפות שונות. לדוגמא, מימוש "חומת-אש" על גבי כרטיס הרשת יכול לבלום התקפות שונות עוד לפני שהגיעו ליחידת העיבוד המרכזית. בנוסף, קוד שאיננו ניתן לשינוי ("צרוב") בכרטיס יכול לפקח על תהליכים שונים ומבני נתונים שונים של מערכת ההפעלה ולהתריע על התקפה פוטנציאלית.
5. <u>הגדלת נפח תקשורת</u> – למרות מהירותם הגדולה של המעבדים המודרניים, נפח התקשרות אותו המעבדים יכולים לעבד הוא מוגבל. מחקר שבוצע לאחרונה בחברת אינטל [25] מראה שעבור חבילות תקשורת הגדולות מ-1KB, נדרש 1Hz לטיפול ב- 1bps. כלומר נדרש מספר לא מבוטל של מעבדים לטיפול בנפחי התקשורת הצפויים (10Gbps ויותר). הורדת פעולות לכרטיסי הרשת תאפשר הורדת העומס מהמעבדים המרכזיים ותשפר את תפוקת המערכת. מערכות המורידות חלק מפרוטוקול TCP לכרטיס כבר קיימות אולם מחקר זה מרחיב את המודל ומאפשר הורדת פונקציונאליות כללית המוגדרת על ידי מפתחי האפליקציות **לכל** כרטיס היקפי.

במחקר המתואר להלן, פותחה מערכת ייחודית המאפשרת למפתח האפליקציה למפות את הקומפוננטות השונות (המרכיבות את האפליקציה) אל אוסף רכיבים היקפיים מיתכנתים עוד בשלב **התכנון**. במודל המוצע, אפליקציות מורכבות מקומפוננטות שונות בעלות ממשק סטנדרטי בעל זיהוי ייחודי במערכת. מודל זה מאפשר שיתוף של קומפוננט על ידי מספר אפליקציות ושימוש חוזר שלהם באפליקציות שונות. אנו חוזים מודל בו מפתחים יוכלו לפתח קומפוננטות חדשות או להורידם בצורה חופשית באינטרנט עבור רכיב היקפי מסוים (לעיתים הקומפוננט יינתן בצורה בינארית ע"י היצרן ואף חתום על ידו עבור התקן מסוים).

מחקר זה מגדיר את האבסטראקציות הנחוצות על מנת לפתח אפליקציה העושה שימוש בקומפוננטות אלה. המחקר מגדיר מהי קומפוננטה, כיצד אחת מגדירה תלות בשנייה, וכיצד אפליקציה מתקשרת איתה. בשלב הרצת האפליקציה, מערכת ההפעלה מבצעת תהליך של "הורדת" הקומפוננטות השונות לרכיבים המיתכנתים. בשלב זה, המערכת נדרשת לבצע התאמה של קומפוננטה לרכיב היעד בצורה פרטנית. פעולה זו עושה שימוש בכלי ההידור והקישור של הכרטיסים ההיקפיים תוך שימוש בפרוטוקול כללי וגנרי.

במסגרת מחקר זה, המודל מומש ונבדק עבור כרטיס רשת עבורו פותחה מערכת הפעלה ייחודית הנקראת: NicOS. מערכת זו שימשה כפלטפורמה לבדיקת התשתית ועל בסיסה מומשו מספר אפליקציות כגון: מחולל חבילות, "חומת-אש", פרוטוקול לסידור הודעות ברשת מקומית ושרת סרטים העושות שימוש בתשתית על מנת להוריד רכיבים לכרטיס הרשת - פעולה שהאיצה באופן משמעותי את ביצועיהן. במהלך פיתוח מערכת ההפעלה אף פותחה שיטה ייחודית לשיפור אלגוריתמי תזמון במערכות מחשב ללא פסיקות. האלגוריתם שפותח מאפשר שיפור התפוקה והקטנת זמן התגובה של תהליכים המתוזמנים במערכת כזו ולמעשה מאפשר שיפור של כל אלגוריתם תזמון ללא פסיקות קיים.

אנו צופים כי מערכות מחשב אישיות מרובות מעבדים יהפכו להיות דבר מן השגרה בעתיד הקרוב. הקהילה האקדמאית ותעשיית התוכנה העולמית ייאלצו לספק כלי פיתוח שונים על מנת לאפשר פיתוח אפליקציות מתקדמות ויעילות העושות שימוש **<u>בכל</u>** המעבדים הנמצאים המערכת המחשוב המקומית. מיקומם הפיסי של המעבדים יהפוך במהרה ללא רלוונטי מנקודת ראותה של מערכת ההפעלה המקומית. מחקר זה הנו אבן דרך במסע לעבר מערכת הפעלה אסימטרית אמיתית שתאפשר שימוש פשוט, יעיל ואינטואיטיבי בכל אמצעי המחשוב העומדים לרשותנו כיום.

An Operating System Specification for Dynamic Code Offloading to Programmable Devices

Thesis for the degree of

DOCTOR of PHILOSOPHY

by

Yaron Weinsberg

SUBMITTED TO THE SENATE OF
THE HEBREW UNIVERSITY OF JERUSALEM

October 2007

This work was carried out under the supervision of:

Prof. Danny Dolev

Acknowledgements

First of all, I would like to deeply thank my thesis advisor, Prof. Danny Dolev that introduced me to the fascinating world of research. Danny's ability to analyze and solve problems differently than most people do, challenged me and made me a better researcher.

Next, I thank Dr. Tal Anker my co-adviser and friend. Tal's indispensable advices and constant support throughout my studies has been a crucial element in this research.

Next, I wish to thank several colleagues and friends: Dr. Pete Wyckoff and Muli Ben-Yehuda, for their detailed and constructive ideas and comments throughout this work. I also feel thankful to Dr. Ophir Holder who was always a good friend and 'informal mentor' during my PhD.

I would also like to thank all the members of the Distributed Algorithms, Networking and Secure Systems Group (DANSS). Specifically I would like to thank Danny Bickson, Shimrit Tzur-David and Dr. Ariel Daliot for their help and support.

Special thanks to my parents Hava and Berco Weinsberg, for their love and encouragement.

Last but not least, I am very grateful to my wife Keren and two wonderful kids: Yuval and Tamaer, for their love and patience during my PhD studies.

Abstract

The constant race for faster and more powerful CPUs is drawing to a close. No longer is it possible to significantly increase the speed of the CPU without paying a crushing penalty in power consumption and production costs. Instead of increasing single thread performance, the industry is turning to multiple CPU threads or cores (such as SMT and CMP) and heterogeneous CPU architectures (such as the Cell Broadband Engine). While this is a step in the right direction, in every modern PC there is a wealth of untapped compute resources. The NIC has a CPU; the disk controller is programmable; some high-end graphics adapters are already more powerful than host CPUs. Our operating systems must let applications tap into these computational resources and make the best use of them.

This dissertation considers the model where applications execute cooperatively in the host processor as well as in device peripherals. In this model, applications can delegate tasks to devices with various architectures and constraints. Using programmable devices has traditionally been very difficult, requiring experienced embedded software designers to implement conceptually simple tasks. Interfacing a new device feature with the host operating system would be performed from scratch and customized for the particular design. The availability of cross-compilation tools and remote debugging environments are making the programming tasks simpler, but integration with the host operating system is still difficult.

This work introduces the concept of an "*offloading layout*" as a new phase in the process of an application development. After designing the application's logic, the programmer will design the offloading layout using a generic set of abstractions. The layout describes the interaction between the application and the offloaded code at various phases, such as deployment, execution and termination.

Today, there is no generic programming model and corresponding runtime support that enables a developer to design the *offloading* aspects of an application. This research involves the design and implementation of a framework to address these challenges.

Contents

Chapter 1

Introduction

Today's modern operating systems (OSs) are complex programs that perform multiple tasks, doing much more than just multiplexing the computer's hardware among applications. An OS provides many of the programming APIs and run-time libraries needed by applications developers. Even the simplest task, such as connecting to a peer host over a network, is performed by user level libraries and complementary kernel runtime support.

State-of-the-art peripheral devices allow one to program the peripheral device and adapt its functionality. For example, modern graphic adapters can perform matrix operations much faster than host CPUs. Today peripheral devices are largely ignored and their increasingly powerful computational capabilities are not being exploited. If peripheral devices could be adapted dynamically to an application's needs, and if their extra computing power could be harnessed to serve the application, bigger, better and more powerful computer systems could be created.

This research considers a model in which applications execute cooperatively and concurrently in host processors and in device peripherals. In this model, applications can *offload* specific tasks to devices to improve the overall performance. Using programmable devices has traditionally been very difficult, requiring experienced embedded software designers to implement conceptually simple tasks. In such cases, interfacing any new device feature with the host operating system would have to be performed from scratch and customized for the particular design. The availability of cross-compilation tools and remote debugging environments are making the programming tasks simpler, but integration with the host operating system is still difficult. The need for new abstractions and tools for programming such heterogeneous systems is apparent.

This research proposes an innovative programming model and runtime support that enables utilization of such devices by providing a generic code offloading framework (called: HYDRA). The framework enables an application developer to design the offloading aspects of the application by specifying an "*offloading layout*", which is enforced by the runtime during application deployment. The framework also provides the necessary development tools and programming constructs for developing such applications.

1.1 Offloading Vs. Onloading

Offloading has been traditionally synonymous with TCP Offload Engine (TOE) devices [18]. Although offloading practices were and still are raising eyebrows, it is agreed that TOE devices perform well for specific types of workloads and applications [45]. The offloading concept can be generalized to any programmable peripheral device and extended to include more than network protocols. For example, file system related functionality such as indexing or searching could be offloaded to a programmable disk controller. Leveraging the proximity between the computational task and the data on which it operates may boost the system's performance and reduce the load on the host processor and memory subsystem. Offloading to several devices at once adds a new dimension to our ability to handle information close to its source with limited involvement of the central CPUs. In particular, expensive memory bus crossings are eliminated.

An offloading adversary will typically claim that although peripheral devices are powerful, today's PCs have several underutilized processors that could be used instead. In response, the following compelling arguments are presneted in favor of offloading:

1. *Memory bottlenecks* — Modern processors have large L2 caches in order to try and minimize cache misses caused by application execution and context swapping. Operations running on peripherals utilize local memory and filter out the information that needs to be brought to and from main memory, hence reduce memory pressure and cache misses on the main processors.

2. *Timeliness guarantees* — Operations running on peripheral devices can benefit from real-time programming paradigms. A peripheral device can provide operation timeliness guarantees that cannot be matched by a general purpose kernel [65].

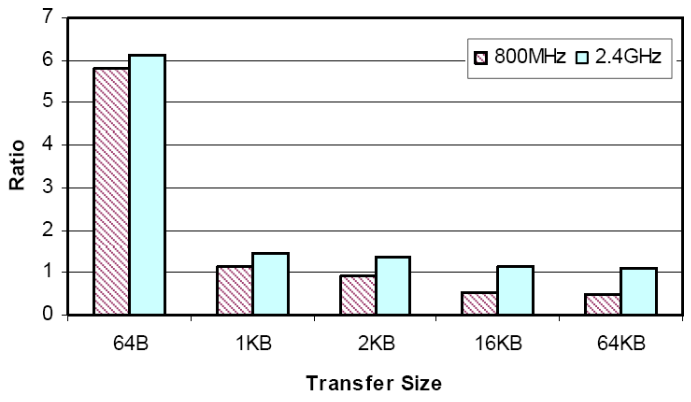
3. *Reduced power consumption* — There is a major effort to reduce the power consumption of modern processors. Some processors support an idle mode with reduced power consumption. By offloading operations to low powered peripherals, we enhance the overall system power consumption (For example, a Pentium 4 2.8 GHz processor consumes 68 W whereas an Intel XScale 600 MHz processor, commonly found in peripheral devices, consumes 0.5 W, two orders of magnitude less).

4. *Security* — partitioning critical code between the host and the peripherals will make it less susceptible to automated attacks. For example, a small watchdog that periodically verifies that the main OS hasn't been tempered with could be run on an offload-capable device. Because it is running in a different environment it can be designed such that automated attacks will be less likely to target it successfully.

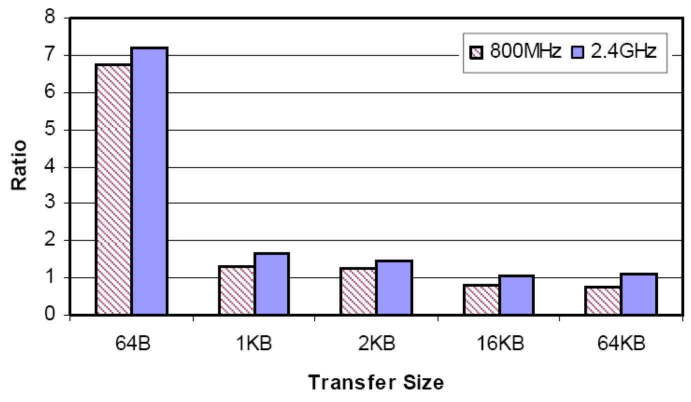
5. *Increased throughput* — Network bandwidth has reached the point where host CPUs can spend all of their cycles just processing network traffic [26]. Specifically, Figure 1.1(a) and Figure 1.1(b) show the GHz/Gbps Ratio in the transmit and receive cases respectively.[1] Although TCP offloading (see Chapter 3) can improve the achieved throughput, it is only one of the potential uses for offloading. This thesis suggests further opportunities in this area.

A recent alternative to offloading has been commonly referred to as “onloading”. Rather than moving functionality to the device, “onloading” proposes using host processors for improving I/O devices' performance. For example the Piglet [46] operating system dedicates one or more host CPUs to provide a “Virtual Device Interface”. Such an interface is directly accessible by user-space applications via shared memory. Although onloading part of the device's functionality to a host processor can yield better performance, eventually the data will need to be transferred between the

[1]These figures appear in [26] and are used with the authors' permission.



(a) GHz/Gbps Transmit Ratio



(b) GHz/Gbps Receive Ratio

Figure 1.1: GHz/Gbps Ratio Validation

host CPU and the device. Such transactions will still incur the known (and sometime unnecessary) overhead at the I/O interconnect.

Another onloading direction has been recently proposed by Intel [53]. The paper proposed to use one of the hosts processors for TCP processing while using several techniques for reducing the protocol computation, data manipulation, and interrupt handling overheads. A step forward in this direction is to fully integrate the network controller with the host CPU [8]. This work presents a simple integrated NIC (SINIC) device that is equivalent to a conventional NIC and is integrated with the host CPU. The SINIC device utilizes zero-copy techniques and was showed to significantly improve the host's throughput.

Even in the presence of "onloading" techniques, history has shown us that applications expand to fill the computational resources available to them. Modern hardware devices, especially high-end devices, often have their own CPUs and memory. Such devices resemble general purpose computer systems, albeit systems that are customized for a specific set of tasks. Operating systems have always been and will continue to be the conduit between the applications and the hardware; we argue that modern operating systems have been remiss in neglecting to provide applications with seamless access to the wealth of computational resources available on peripheral devices.

1.2 The Future Of Offloading

In the near future a handful of computing resources will be available in any home PC. Treating these computing resources as first class citizens and offloading computation and functionalities to them wherever and whenever possible will enable development of high performance applications that will benefit from the unique capabilities of each resource. This section briefly presents some of the potential fields that will benefit from the offloading capabilities.

1.2.1 Virtualization

Rapidly improving virtualization technologies allow one to run multiple OSs simultaneously on one physical machine, as “virtual machines”. Running multiple operating systems on the same physical machine places considerable demand on the “host software”, due to the need to multiplex the physical resources among virtual machines. Offloading computation to peripheral resources offers several opportunities to alleviate this burden.

Current virtualization technologies prevent virtual machines from directly accessing I/O devices due to functional, security and isolation concerns. From a functional point of view, nearly all current devices are fundamentally designed to be accessed by a single entity (e.g., devices have a single register window). From a security and isolation point of view, current server chipsets allow devices to DMA anywhere in physical memory, since the assumption is that they are being programmed by a trusted entity. If an *untrusted* virtual machine could directly program a device, it could program it to DMA anywhere in memory, including on top of the hypervisor or other virtual machines, thereby bypassing the hypervisor’s isolation guarantees.

Due to the above limitations, all I/O device accesses by virtual machines are either multiplexed or emulated by the hypervisor or a service OS running on the host CPUs, which then perform the real I/O to the physical device. Such an architecture incurs heavy performance costs when compared with direct device access by virtual machines.

The introduction of IOMMUs [6] and self-virtualizing devices for virtualization should alleviate the security and isolation concerns mentioned above; the functional limitations of current devices could be overcome by utilizing programmable devices. Offload-capable devices could perform more efficiently some of the tasks that are executed today by the host CPUs, such as multiplexing incoming network packets directly to the destination virtual machine. In the event that device emulation is needed because the virtual machine does not have a driver for the physical device, an offload-capable device could emulate a virtual device directly on the physical device.

1.2.2 Gaming

Hardcore PC gamers live and die upon squeezing every drop of performance out of their hardware. The graphics and networking technology presented in Section 3.4 enhances the gaming experience. For example, the Killer NIC [1] completely takes over all networking tasks traditionally handled by the OS and processed by the CPU, effectively bypassing the OS networking stack. Since the NIC still needs to pass the packets to the GPU through the host processor, a generic offloading framework may further improve the achieved performance by enabling direct interaction of the host software and the GPU, with minimal host CPU involvement, thereby increasing availability of main CPU cycles for manipulating the more advanced scenes.

1.2.3 Advanced Storage Services

Programmability support that will soon be offered by advanced disk controllers and external storage controllers will open new possibilities for implementing advanced storage services directly inside the disk or controller. One example is the Diamond system [35] that employs “early discard”, which involves rejecting irrelevant data as early in the pipeline as possible. Diamond applications can install filters at the active disk for reducing data transfer.

In general, programmable disks or controllers will provide an opportunity to run I/O intensive computations efficiently by running them closer to the data. Potential applications include content indexing and searching, virus scanning, storage backup, mirroring, snapshots and continuous data protection.

1.2.4 Accelerating Distributed Applications

An important aspect of this research is to develop basic distributed protocols that take advantage of the newly developed framework, reducing currently accepted inherent uncertainty of distributed systems, and increasing robustness and security of the resulting systems. To illuminate some aspects of the significance of the new approach the next few sections discuss some traditional distributed computing approaches that can benefit from the offloading capability offered by the proposed framework.

Network Oriented Components

Distributed applications operate by interchanging messages among nodes. The message exchange networking protocols are potential candidates for offloading. For example, the reliable broadcast service that ensures that all hosts in a group of nodes deliver the same set of messages to the application layer can be offloaded to the networking device. This service can be used as a building block to construct value-added multicast services, such as agreement and total ordering, or it can be utilized to support applications that involve groups of cooperating hosts.

The network components can be also used for various functions like: early filtering of data, identifying patterns in the message flow that indicate possible attacks [82], consistency verification of the transmitted messages, and possibly signing or authenticating the message source or target.

Cluster Synchronization

Real-time guarantees can be implemented on programmable peripheral devices [73] and used as a building block for a variety of distributed applications as exemplified by the work by Verissimo et al. [68]. Having such a timely component significantly simplifies the design of real-time algorithms. This component is an ideal candidate for offloading, as it exports a simple interface that is ideal for a programmable clock, network card, or encryption engine. Once we are provisioned with real-time guarantees with smaller uncertainty, we can further increase the ability of the application to cope with transient or permanent failures.

Virtual Synchrony

The virtual synchrony model [10] offers strong communication guarantees required by applications such as replicated database systems [38,34]. The overhead involved can be drastically reduced, and performance correspondingly increased, by offloading the critical components to the networking card. Node failures may be detected faster and more reliably. Virtual synchrony is critical in ensuring the consistency of the views of the system at the various participants. This consistency is a key component in increasing the robustness of the system and in limiting the ability of an outside entity to jeopardize the system's objectives.

Byzantine Consensus Protocols

When building secure replicated systems, the replicas have to coordinate updates using Byzantine Consensus [39]. These protocols are complicated and message intensive. Offloading them to a network device would simplify application development and improve their performance.

Self-Stabilizing Protocols

These protocols are designed to return a system to a normal functioning, irrespective of the severity and nature of transient failures, as long as there is a sufficiently long time interval during which a large enough portion of the system behaves correctly. Offloading some of the functionality can significantly reduce the convergence time. For example, due to the higher reliability of the NIC, the self-stabilizing protocol may significantly decrease the time required to trust the coherence of the received messages by verifying them with the NIC.

Until recently dealing with worst case failure and self-stabilization were considered infeasible. The known protocols required exponential convergence time [23, 24]. Recent results [19, 20] indicate that convergence can become linear, though the protocols are still involved and use complicated Byzantine agreement modules. Taking the advantage of programmable devices may assist in developing efficient distributed self-stabilizing protocols that withstand the permanent presence of on-going faults. Moreover, when cluster synchronization (as suggested above) is available, the protocols can be drastically simplified.

Cryptographic Modules

Many modern secure computing modules require on-line and consistent exchange of messages among parties. They also require renewal of signature keys and coordination of sharing of secrets. Having secure modules residing on independent devices can further simplify and boost multi-party-computations and other secure computations. Offloading parts of the security services of a general purpose OS to such devices can significantly improve their performance, reliability and may also reduce the probability of attack due to their isolation properties. Moreover, such a card can be in charge of some critical functions such as the certification authority (CA).

Secure Fingerpointing

One interesting application enabled by offloading is run-time checking of global behavior of distributed applications. Thus, rather than having an offloaded component on a peripheral device checking the local behavior of an application on the main CPU, such a component on a networked device can communicate with other such components at other nodes to check global behavior. For example, consider a peer-to-peer video streaming service. Nodes in such a service may exhibit rational behavior by not forwarding video fragments upon receipt. Downstream nodes may complain about this behavior, possibly resulting in removal of the upstream node. Unfortunately, it may be the rational behavior of the downstream node that lies and falsely accuses the upstream node in order to get closer to the source of the video. Other peers have no way of verifying whether it is the upstream node or the downstream node that is behaving badly. Offloaded components on the network devices of the two nodes could easily tell what is going on, however.

1.2.5 Isolation of Device Drivers

Reliability is now the greatest challenge for computer systems research. Considerable resources are invested by major operating system vendors for systematically auditing Windows and Linux device-driver code for flaws. A recent study performed in Stanford University found that more than 50% of the Linux OS bugs, appear at device drivers [15]. In Windows XP, drivers account for 85% of recently reported failures.

One of the hypothesis explaining this phenomena is that device drivers are typically written by device vendors which do not necessarily have the same level of understanding of OS internals as the OS developers. Device driver bugs are often fatal; since the number of device drivers in modern operating systems is enormous, there is an apparent need to improve device driver reliability.

One possible approach is to isolate device drivers in their own environment. For example, the Nooks project [63] isolates device drivers by using various techniques such as kernel wrapping, virtual memory protection and different privilege levels. However, such isolation incurs significant overhead, especially when the CPU is already saturated (for example, in the Nooks kHTTPd benchmark, the overhead was nearly 60%).

By utilizing programmable devices and a generic offloading framework like the one presented in this dissertation, one could provide yet another level of isolation, by offloading some or all of the driver code to run on the device's CPUs.

1.3 Dissertation Outline

The rest of the dissertation is organized as follows: Chapter 2 discusses the motivation for this work as well as the challenges. Chapter 3 discusses the state-of-the-art in co-processing technologies and specifically describes the related work concerning offloading. Chapter 4 describes the framework's programming model and Chapter 5 discusses the realization of this model including a detailed description of its design. Chapter 6 presents a Network Interface Card Operating System (NICOS) developed as a platform for evaluating the proposed offloading framework. During the development of the operating system, an innovative scheduling algorithm has been designed, implemented and evaluated. This algorithm is also discussed in this chapter. Chapter 7 presents a mathematical approach for presenting complex offloading layout graphs. Chapter 8 presents an evaluation of the proposed framework, and includes qualitative and quantitative results. Chapter 9 presents our conclusions and points to future work.

Chapter 2

Motivation

This section begins with the motivation for developing an offloading framework. We present the requirements from an offloading framework and some of the challenges inherent in offloading and in creating an offloading framework in particular. We then present a simple multimedia application, called TiVoPC, which serves as a motivating example. We provide further details regarding it in Section 8.1.

2.1 Framework's Motivation

Offloading code to a programmable device today is a manual, tedious process, rife with opportunities to reinvent a square wheel. Offloading stand-alone code is difficult; offloading a software component that is part of a larger system with complex interdependencies much more so.

Offloading code to a programmable device requires the following (manual) steps:

- Write it, but do it with the specific constraints of the target environment in mind: does it have an MMU? What sort of run-time support does the device have? Does it support dynamic memory allocation? Is there a toolchain that targets that device for the programmer's preferred language and environment?
- Compile and link it, using a device-specific toolchain. Some of the device-specific aspects mentioned previously might be handled by the toolchain. Linking is usually done with the

device's run-time support libraries, which constrains the programmer to only using an API for the particular device.

- Deploy it on the device. Each device has its own process of transferring the code from an annotated area in host memory to the device, such as through a firmware update.

Additionally, writing offloaded code presents the following challenges:

- There is a steep learning curve. The programmer needs to be acquainted with all the relevant hardware specifications and the relevant SDKs. Additionally, programming a device typically also requires kernel level developing skills including writing device drivers.
- It requires embedded development skills. Usually, it will take an experienced embedded engineer to develop an efficient, stable and robust system.
- It requires dealing with performance issues. While offloading code to a device has numerous advantages, it also has certain disadvantages, e.g., communicating with code running on the host CPU becomes more expensive since the offloaded code is executed in a different hardware domain. This makes getting inter-component information transfer working correctly and efficiently tricky.
- The bulk of the work needs to be redone for every new device.

An offloading framework should facilitate and automate as many of the previous steps as possible. It should also ease the aforementioned challenges of writing offloaded code. The holy grail is for the programmer to be completely unaware of the fact that parts of the system she is writing will be running on a programmable device. To achieve these goals, an offloading framework must meet the following requirements:

1. It should not require the programmer to learn a new language or a new environment.
2. It should abstract the specific details of given devices as much as possible, so that the framework will handle the adaptation of the offloaded code to a specific offload target, rather than the programmer. This includes the specific hardware details as well as the specific run-time support provided for the device. Similar classes of devices (e.g., NIC, or GPUs) provide

roughly the same functionality or capabilities, albeit in different ways and using different interfaces. The offload framework should abstract these device specific details behind a common abstraction layer.

3. It should ease deployment, by deciding when and where to deploy a given component, as well as facilitating communication of the deployed component with the rest of the components, whether they are running on the main CPU, on the same device or on a different device.

2.2 TiVoPC: A Motivating Example

In order to provide insight into the usefulness of offloading to multiple devices using the framework, we now present a sample application, which we call TiVoPC. We show that with the right set of abstractions and development tools, offloading becomes feasible and desirable.

The TiVoPC is a software implementation of the commercial TV appliance Tivo [64]. A classical Tivo appliance is a box that allows for digitally recording all of one's favorite TV shows, and enables playback of them at a later time. Our implementation of the Tivo appliance provides a selected subset of the Tivo features. Specifically, we provide online-recording while watching a media stream and support its playback at a later time. A typical user-space software implementation of such an appliance would require the following components listed in Table 2.1.

Component	**Description**
GUI	Provides the viewing area and user controls (play, pause, rewind and resume).
Streamer	Processes the media stream (either from network or storage).
Decoder	Decodes the MPEG media stream.
Display	Displays the movie on screen.
File	Reads/Writes previously stored data from storage.

Table 2.1: TiVoPC Components Outline

When analyzing a TiVoPC operation, one can see that a major part of the application logic is invested in transferring packets from one I/O device to another. Specifically, the Streamer component transfers each received packet to the File component, in order to support a later playback, and to the Decoder component. The decoding component hands a decoded frame to the Display Component, which transfers the raw video frames to the graphics subsystem.

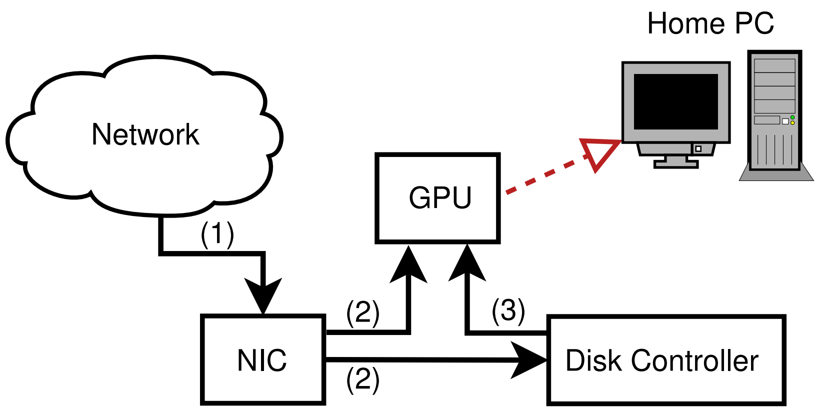


Figure 2.1: TiVoPC Data Flow

In order to demonstrate the use of our framework we have implemented a version of the TiVoPC application that uses multiple peripheral devices. In Figure 2.1, the resulting data flow of the offload-aware TiVoPC application is presented. Once a packet is received at the NIC, it is directly transferred to both the GPU and the disk controller.[1] A decoder component running on the the GPU can directly decode the MPEG stream and transfer each frame to the GPU's internal framebuffer, making it appear in the GUI window without involving the host CPU at all. In case a user wishes to replay the stored media, a Streamer component running on the disk-controller will transfer previously stored packets to the Decoder. Section 8.1 provides the full details of the implementation.

2.3 Research Objectives

Today, there is no generic programming model and corresponding runtime support that enables a developer to design the offloading aspects of an application. The development of this approach requires to revisit many traditional aspects of distributed operating systems, shared memory algorithms, advanced compilation techniques and distributed and parallel algorithms.

The inherent conflict between the heterogeneousness nature of programmable peripheral devices and our requirement to provide a generic framework and simple programming interface, introduces several challenges:

- *Components and Device mapping –* Components have predefined properties and runtime

[1]Note that if the bus architecture allows it (i.e., PCIe), this packet could be transferred in a single bus transaction.

assumptions. Develop a way to identify the possible matching between the devices' capabilities and the component's requirements.

- *Device and Component Reuse* – Devices are resource constrained. Devise a scheme for efficiently reusing both the component and the resources at target devices.

- *Operating System extensions* – There is a need to design specific OS algorithms for optimizing the application's offloaded code performance. For example, new scheduling algorithms with real time guarantees, dynamic code loaders, memory management and buffers reuse.

- *Dynamic Offloading* – Components can be given as binaries or open source. Devise a scheme for dynamic compilation and/or offloading of such components to different devices with different architectures.

- *Dynamic Conflict Resolution* – Different applications have different offloading requirements. Since devices are shared by multiple applications, an online algorithm for scheduling and for optimal placement needs to be developed.

- *Communication Model and Buffer Management* – The framework needs to address the various communication flows, between the application and its components and among components, potentially residing at different devices. A major challenge is to minimize the communication overhead, for example using a zero copy semantics.

- *Security* – Improve the system's security using hardware devices without increasing the system's vulnerability.

- *Failures and Failover* – Increase the system's robustness by making use of the devices and by enabling self-healing properties.

One aspect of our research includes the design and implementation of a framework to address these challenges. Section 4 introduces the concept of an "offloading layout" as a new phase in the process of an application development. After designing the application's logic, the programmer will design the offloading layout using a generic set of abstractions. The layout describes the interaction between the application and the offloaded code at various phases, such as deployment, execution and termination.

Chapter 3

Related Work

Offloaded applications have been designed for particular needs in the past using specific devices. Some of this work has led to the availability near-commodity products. This section describes the state-of-the-art in offloading research, ordered by its relevance to this work.

3.1 Storage Offload

Object Storage Devices (OSD) came from a research project called Active Disks from CMU [55, 56] and are approaching standardization by the ANSI T10 group [72]. OSD is a protocol that defines higher-level methods for the creation, writing, reading and deleting of data objects on a disk. Implementing OSD requires a high degree of processing capability at the disk controllers or the devices themselves and can offer the potential for extension.

One example of a storage-specific extension is the Diamond system [35]. Unlike traditional architectures for exhaustive search in databases, where all of the data must be shipped from the disk to the host computer, the Diamond architecture employs “early discard.” Early discard is the idea of rejecting irrelevant data as early in the pipeline as possible. By exploiting active storage devices, one can eliminate a large fraction of the data before it is sent over the interconnect to the host. Diamond applications can install filters at the active disk for eliminating data.

3.2 Network Offload

One of the more fruitful areas for exploiting programmable devices is in the area of networking. As wire speeds increase and demand extensive host processing power, moving some of the work to the network card becomes an attractive alternative. Although previous research have also considered using programmable components to accelerate network processing in specific situations [43, 29]. This research goal is to enable more general access to programmable components for arbitrary networking or other computing or I/O tasks.

3.2.1 Spine

Spine [25] is a safe execution environment, derived from the SPIN operating system [7] that is appropriate for programmable network interface cards. Spine enables the installation of user handlers, written in Modula-3, on the NIC. Applications and extensions communicate via a message-passing model based on Active Messages [69]. Although Spine enables the extension of host applications to use NIC resources it has several major limitations. First, since all extensions are executed when an event occurs, building stand-alone applications for the NIC is difficult. Even for event-driven applications, the developer is enforced to dissect the application logic to create a set of handlers. Second, Spine's runtime does not support the deployment process of handlers or provide a way to design the offloading aspects of the host application.

3.2.2 Arsenic and EMP

Arsenic [50] is a Gigabit Ethernet NIC that exports an extended interface to the host operating system. Unlike conventional adaptors, it implements some of the protection and multiplexing functions traditionally performed by the operating system. This enables applications to directly access the NIC, thus bypassing the OS. The Ethernet Message Passing (EMP) [58] system is a zero-copy and OS-bypass messaging layer for Gigabit Ethernet. EMP protocol processing is done at the NIC and a host application (usually through an MPI library) can directly manipulate the NIC. Arsenic and EMP provide very low message latency and high throughput but are very task-specific and lack the support for generic offloading or host application integration.

3.2.3 TCP Offload Engines (TOE)

TOE [18] is a technique used to move some of the TCP/IP network stack processing out of the main host and into a network card. Commercial NICs that support TOE extensions exist but lack any open standard specification. Typically, these devices include several on-board programmable processors that are only programmable by the device manufacturer. While TOE technology has been available for years and continues to gain popularity, it has been less than successful from a deployment standpoint. TOE only targets the TCP protocol, thus, user extensions are out of its scope. Practical concerns such as the inability to modify TOE behavior for evolving TCP protocol changes or to implement complex firewalls also limit the utility of such devices.

Microsoft's support for TOE devices is supported through the "Chimney Offload Architecture" for Windows [17]. Chimney provides a standard interface for TOE devices and enables the offload of the TCP/IP *data path* to the target device. Other protocols such as DHCP, RIP, IGMP, and ARP are implemented within the traditional TCP/IP networking stack.

Linux kernel does not officially plan to support TCP offload engines. The networking maintainers believe that TOE support, inside the Linux kernel, may cause enormous maintenance problems. For example, testing is problematic since the hardware firmware sources are proprietary.

Other approaches for reducing network processing overheads that are based on TOE devices are possible as well. iWARP [52] is an approach that takes advantage of Remote Direct Memory Access (RDMA) [51] and processor offload to increase throughput and reduce host overhead. iWARP network cards include TOEs and other functionality needed to implement the higher-layer protocols.

3.3 Computation Offload

Specific devices to assist a host processor with some of its computational burdens have existed for many years and seem to be experiencing a recent resurgence.

Field-Programmable Gate Arrays (FPGAs) can be used as CPU accelerators that can be plugged directly into a standard processor socket or as add-in PCI cards for supercomputer systems. For instance, the DRC coprocessor [48] is an FPGA device that plugs directly into a processor socket in an Opteron system. Placing the DRC within the CPU fabric accelerates the communication with the host CPUs. Other examples are the iPath Infiniband adapter that plugs directly into an AMD HT socket and IBM's system-z cryptography [36] coprocessors. These cards assist various cryptographic functions (e.g., DES, Triple DES, hashing etc.). Offloading parts of the security services of a general purpose OS to such devices can significantly improve their performance and reliability, and may also reduce the probability of hacker attacks due to their isolation properties. Moreover, such a card can serve as a certified and independent authority.

Each FPGA vendor provides varying level of support for the development of host applications and device programs ranging from a single high-level language and auto-generating compilers down to explicit device gate design. What is lacking in FPGA development is any generic interface or commonality that would enable applications to run on platforms other than where they were developed. Also the communication models for FPGAs are typically primitive compared to the networking and storage examples described above.

3.4 Graphics Offload

The recent boost in GPU technologies have made them more powerful than ever. Compared to the CPU, GPU performance has been increasing at a much faster rate than CPU performance. The work presented in [30], uses an NVIDIA 7800GT GPU for sorting database records. The GPU's computing power and the high-bandwidth GPU memory interface enable their system to achieve better performance than the CPU-based algorithms.

Other recently developed GPUs include ClearSpeed's accelerator [16] which provides ~ 50 GFLOPS sustained performance and accelerates most of the standard math libraries; and the Ageia physics processing unit [49] that can be used for optimizing game physics.

3.5 Onloading

The Simultaneous Multi-Threading (SMT) [66], also known as hyper-threading, and the Chip Multiprocessing (CMP) architectures have already been adopted as the architecture for processors of the future. Unlike Symmetrical Multi Processing (SMP), where host CPUs are completely homogeneous, the SMT and CMP architectures are taking small steps toward heterogeneity.

The proliferation of host level processing cores have motivated researchers to explore other alternatives for offloading. As discussed in Section 1.1 a recent alternative to offloading has been commonly referred to as “onloading”. The idea presented in details in the Piglet [46] operating system has been realized in the work of Greg et al. [54]. The Embedded Transport Acceleration (ETA) approach dedicate one or more processors for executing the TCP/IP network stack code. ETA achieves the same performance as a regular Linux machine but with a reduced CPU utilization. The paper shows that the dedicated processor becomes a bottleneck in the system due to expensive memory operations. The paper implies that by following a full offloading approach such limitations may be eliminated which agrees with this research motivation.

3.6 Related Frameworks

3.6.1 FlowOS

FlowOS [12] proposes an architecture that removes the host’s memory subsystem and CPU from the critical data path. The main role of the OS is to manage the data-flow between different peripheral devices and to schedule the flows between different applications. Although FlowOS does not provide an offloading framework nor a programming model for creating offload-aware applications, the proposed flow abstraction can further extend this research. By defining a “flow” overlay that spans several offloaded applications, one can guarantee the required QoS for a specific application.

3.6.2 FarGo and FarGo-DA

Although not dealing with offloading, FarGo [33, 32] and FarGo-DA [74] propose a programming model that enables a developer to program relocation and disconnection semantics in a separate phase during the application development cycle. The basic assumption for their work is that the application is fully comprised of a set of components that are tagged by a specific interface (called: *Complet*). The components are hosted in a virtual machine and can migrate to a remote VM using marshaling and unmarshaling mechanisms (much like in the RPC [44, 9], RMI [61, 62], CORBA [60], DCOM [11] or WebService [80] models). Our framework extends this model by defining an “offloading-layout” that is used to define the offloading aspects of the application.

3.7 Summary

In this chapter we surveyed the existing body of work in the area of offloading, specifically storage offload, network offload and GPU offload. The idea itself is not particularly new. Systems that split the workload between a general-purpose processor and specialized coprocessors, have been around for years. Many of these systems started with the goal of improving the performance of a specific application. Yet, providing the basic primitives in order to program the offloading semantics at the application level is an issue that was not at the focus of any other research that we know of. Moreover, the development of a special programming model and examination of the specific system support which is required for realizing such a model is a unique goal of our research.

Chapter 4

Programming Model

The programming model provided by HYDRA enables an application developer to design offload-aware applications (henceforth, OA-applications). Such applications can utilize any available computing resource that offers programmability support. The model proposes an object-oriented methodology for developing such applications. Developers use a set of special components called Offcodes. An Offcode is a component that contains a state, defines a unique interface and is executed by a dedicated thread.

Communication between Offcodes is facilitated by communication channels with various communication properties as will be presented in Section 4.2. The programming model is divided into two coupled facets:

1. *Application logic programming* — This is the mechanism of designing the basic logic of the application. Offcodes are provided as a set of reusable components from the vendor or custom made by the developer.

2. *Offload layout programming* — This task defines the mapping between components and peripheral devices, both in software and hardware. It also sets offloading priorities and channel characteristics between Offcodes and the host.

Programming the application logic should resemble programming a regular application while programming the layout should affect the application logic as little as possible. The developer is encouraged to reuse Offcodes that are provided as a set of components from the vendor or custom made by the developer. The process of placing Offcodes at the peripheral devices involves defining the mapping between components and peripheral devices, both in software and hardware as will be discussed in Section 4.4.

4.1 Offcodes

We envision openly accessed libraries of Offcodes that are provided as source code, or as object files that can be linked together with the target device's firmware. An Offcode is described by an Offcode Description File (ODF) that uses XML to describe the supported interfaces, dependencies on other Offcodes, and the target device's hardware and software requirements. A detailed description of the ODF file and the deployment process is given in Section 4.3.

An Offcode can implement multiple interfaces, each of which contains a set of methods that perform some behavior. Each interface is uniquely identified by a Globally Unique Identifier (GUID) and is also described by the ODF file using the standard Web Service Definition Language [70] (WSDL). An offload-aware application communicates with an Offcode using an abstraction called a *Channel*. An Offcode object file implements only one Offcode and it has a GUID that is unique across all Offcodes. All Offcodes implement a common interface (*IOffcode*) that is used by the runtime to instantiate the Offcode and to obtain a specific Offcode's interface.

4.1.1 Offcode Creation

Offcodes are created by an OA application by calling the runtime *CreateOffcode* API. The method uses the Offcode's ODF file in order to construct an Offcode dependency graph, called the offloading-layout graph, that is used for offloading the OA-applications' Offcodes. Section 4.3 details the mechanism used for the mapping of Offcodes to their respective devices. Once the Offcode is constructed at the target device, it is initialized and executed by the HYDRA runtime. Offcode initialization is performed in two phases. First, the *Initialize* method is called and the Offcode acquires its *local* resources. Since peer Offcodes may have not been offloaded yet, the Offcode can access local resources only. Once all the related Offcodes have been offloaded, the *StartOffcode* method is called. At this point inter-offcode communication is facilitated.

Figure 4.1 presents an Offcode deployment process which is executed by the runtime. The OA-Application running on the host creates a single Offcode α that requires a second Offcode β. Since the Offcode is automatically created, the runtime constructs an offloading-layout graph (Section 4.3) and performs the actual offloading process. The figure illustrates the scenario where α is offloaded before β, but it can also be the opposite. This is why the Offcodes are only allowed to communicate with each other after the *StartOffcode* method is invoked.

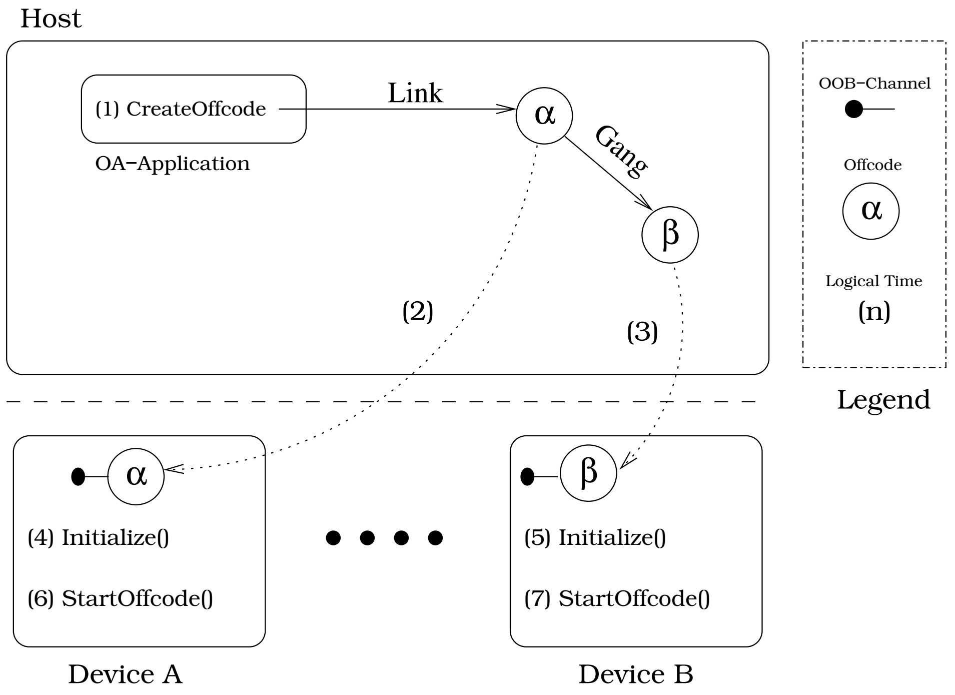


Figure 4.1: Offcode Deployment

4.1.2 Offcode URL

An Offcode is uniquely identified by an Offcode URL. The URL consists of four parts: the host, the device's physical address, the hardware identifier and an Offcode's binding name that is unique per device. The physical address and the hardware identifier uniquely identify the target device. Figure 4.2 presents an Offcode's URL format and a sample URL for some PCI device. A PCI device is physically addressable by a bus number (8 bits), a device number (5 bits) and a function number (3 bits). The hardware identifier is further identified by a 32 bit signature that includes the vendor identifier and the device identifier.

```
[host]:/[physical-address]/[hardware identifier]/[binding-name]

Example:
------
The Hydra runtime offcode on the Netgear GA-620T (TigonII chipset)
device is identified by the string:
   localhost:/pci/00/11/1385/620A/Hydra.Runtime
```

Figure 4.2: Offcode URL Format

Note that the full Offcode's identifier is automatically created by the runtime once the Offcode's target device is determined (see Section 4.4). A developer typically uses the Offcode's binding name in order to communicate with a peer Offcode.

4.1.3 Offcode Attributes

Once an Offcode has been explicitly created, a set of attributes can be applied to it. The programming API enables a developer to handle Offcode attributes using the *setAttribute* and *getAttribute* methods. These methods take two arguments: an attribute identifier and a value.

HYDRA currently supports the following attributes:

OBSOLETE_TIME – The attribute enables a developer to determine the amount of time an Offcode should "live". The time is measured relative to the Offcode's offloading time. The attribute is usually set for short-lived (or temporary) Offcodes.

WATCHDOG_TIME – The attribute defines the invocation frequency that *must* be maintained for a given Offcode. The runtime automatically disposes Offcodes that have not been responded for more than the given watchdog time period. The attribute facilitates an application level keep-alive mechanism.

OFFLOAD_PRIORITY – The attribute sets the offload priority for the created Offcode. HYDRA currently support three priorities: PRI_LOW, PRI_NORMAL and PRI_HIGH. During application deployment, the offloading process will be executed according to the given priorities.

4.1.4 Offcode Invocation

HYDRA provides two ways to invoke an Offcode: transparently and manually. Achieving syntactic transparency for Offcode invocation requires the use of some “proxy” element that has a similar interface as the target Offcode. When a user creates an Offcode, a proxy object is created and loaded into user-space. The proxy’s job is to perform marshaling (serialization) and unmarshaling (deserialization) of the methods arguments and the returned values, prior/after the method invocation [57].

All interface methods return a *Call* object that contains the relevant method information including the serialized input parameters. Once a *Call* object is obtained, it can be sent to a target device (or several devices) by using a connected channel. The manual invocation scheme consists of manually creating the *Call* object, and using a custom encoder to marshal arguments and invoke the channels’ methods.

4.1.5 The Call Object

A *Call* object is a data structure that encapsulates the relevant information needed to invoke a target Offcode’s method. A *Call* object contains an input buffer with a fixed length header describing the target method identifier, the input buffer encoding and the buffer’s length. According to a given encoding scheme, the buffer is processed by the target method (or proxy). For Methods that return a value, the *Call* object also contains an output buffer. Section 5.3 provides further details regarding the *Call* object.

4.1.6 Call’s Encoding

HYDRA provides a generic encoding scheme that enables a developer to choose a unique encoding per method. A developer can use a custom encoder or a standard one. Many formats and open source encoders are available. SOAP [79, 42] and XML-RPC [81] are simple formats which are widely used. Although some will argue that specialized protocols will be more efficient, in many cases they turn out to be equally or more complicated and costly. Previous work [22,27,31] showed that the overhead imposed by the XML-RPC protocol is negligible (only 16KB in [31]) and the resulting performance is high. Section 5.4 provides further details regarding HYDRA's supported encodings.

4.1.7 Pseudo Offcodes vs. User Offcodes

We distinguish between pseudo Offcodes and user Offcodes. Pseudo Offcodes are runtime components that happen to be implemented as Offcodes, but not written by the user for a particular application. Reasons to do this are because these components export well-defined interfaces, or because of a desire to reduce the processing time for dynamic loading of user Offcodes. By requiring that user Offcodes interact with the device's OS via pseudo Offcodes, we can minimize the required processing of undefined references of an Offcode binary while installing it at the target device. One example for a pseudo Offcode is the "Hydra.Runtime" that provides the runtime functionality through a well defined interface. The runtime's *GetOffCode* method enables a user Offcode to get an interface to any Offcode currently registered at the runtime by providing it the Offcode's GUID. Another example is the "Hydra.Heap" Offcode, which provides an interface to the OS memory routines.

4.2 Channels

Offcodes communicate with each other and with the host application by communication channels. Channels are bidirectional pathways that can be connected between two endpoints, or connectionless when only attached to one endpoint. A channel can be considered as a transport mechanism used for communicating with Offcodes (analogous to the OS sockets abstractions used for networking).

4.2.1 Out-Of-Band Channel

The runtime assigns a default connectionless channel, called the *Out-Of-Band Channel (OOB-channel)* for every OA-application and Offcode. The OOB-channel is identified by a single endpoint used to communicate with the Offcode without the need to construct a connected channel, such as for initialization and control traffic that is not performance critical. The OOB-channel is the default communication mechanism between peer Offcodes and between Offcodes and OA-applications. The OOB-channel is usually used to notify the Offcode regarding management events and availability of other channels. For example, assume that an OA-application communicates with Offcode α using the default OOB-channel. Once the OA-Application creates a specialized channel (see Section 4.2.2) and attaches α to it, the runtime at the target device implicitly creates a corresponding endpoint and notifies α using α's OOB-channel. Once α is notified, it can start listening for requests on the new channel. Figure 4.3 presents the code needed for obtaining the OOB-channel for a sample Offcode.

```
Runtime *rt = GetRuntime();
IChannel *oob = rt->v->GetOOBChannel(rt,"Hydra.net.utils.Socket"
                                     &IID_CHANNEL);
```

Figure 4.3: Obtaining the OOB-channel

4.2.2 Specialized Channel

The OOB-channel can be used for simple data transfer between the application and Offcodes and among Offcodes. For high performance communication, a specialized channel that is tailored to the needs of the application and the Offcode would be created. Enabling a specialized channel is performed in two steps. First, the channel creator determines the channel characteristics and creates its own endpoint of the channel. Second, the creator attaches an Offcode to the channel. This action implicitly constructs the second endpoint at the target device, and notifies the Offcode about the newly available channel. Once the channel is connected, the channel's API can be used for communication. The channel API contains typical operations to read, write and poll. The channel API also supports registration of a dispatch handler that is invoked each time the channel has a new request.

Creating a channel involves configuring the channel type, synchronization requirements and buffer management policy. A channel can be of type *Unicast*, that can only interconnect two Offcodes, or *Multicast*, that can interconnect more than two Offcodes. A channel can be either sequential (synchronized) allowing one invocation at a time or parallel (un-synchronized). A channel can be either unreliable or reliable, where the latter type is careful not to drop messages even though buffer descriptors are not available. Note that a multicast channel can utilize hardware features, if available, to send a single request to multiple recipients simultaneously.

Figure 4.4 presents the typical sequence of operations required to initialize a channel and connect it to a specific device. In this code, a reliable unicast channel is constructed with a zero-copy policy for read/write and sequential synchronization guarantees. A callback handler is then installed at the OA-application side of the channel. The corresponding handler is invoked by the runtime whenever data is available on the channel, as opposed to requiring the application to poll. Connecting an Offcode to a previously created channel is easily performed by calling the channel's *ConnectOffCode* method which takes the target Offcode reference as a parameter.

```
/* get our runtime and create the Offcode */
Runtime *rt = GetRuntime();
IOffcode *ocode=rt->v->CreateOffcode(rt,"/offcodes/checksum.odf",
                                        &IID_Checksum);

/* get the channel executive */
ChannelExecutive *exec;
ErrorCode res=rt->v->GetOffCode(rt,"Hydra.ChannelExecutive",
                                   &IID_ChannelExecutive,
                                   &exec);

/* set up the channel */
ChannelConfig config;
config.type = UNICAST_CHANNEL | RELIABLE_CHANNEL;
config.sync = SYNC_SEQUENTIAL;
config.buffering = DIRECT_READ | DIRECT_WRITE;
config.targetDevice = ocode->v->GetDeviceAddr(ocode);

/* create the channel to our target */
Channel *channel;
channel = exec->v->CreateChannel(exec, &config);

/* install a callback handler */
channel->v->InstallCallHandler(channel, MyHandler);
```

Figure 4.4: Creating a Channel

4.3 Offcode Manifesto

An Offcode manifesto is the means by which an Offcode defines its dependencies on peer Offcodes and its requirements from the target device and software environment.

The manifesto is realized in an Offcode Description File (ODF). An ODF contains three parts: the first part describes the structure of the Offcode's package, containing the binding name of the Offcode at the target device, and the Offcode's supported interfaces. The Offcode's interfaces are typically described by a standard WSDL [70] file. Figure 4.5 presents a typical import section defined in an Offcode's ODF.

```
<ocode>
 <!- ocode package info ->
 <package>
  <bindname>Hydra.net.utils.Socket</bindname>
  <GUID>7070714</GUID>

  <interface>
    <!- WSDL interface specification ->
    <include>/offcodes/socket.wsdl</include>
  </interface>
 </package>
```

Figure 4.5: ODF - Part I

The binding name identifies the Offcode at the target device and it is used in the various HYDRA APIs to identify the Offcode.

```
 <!- ocode dependencies ->
 <sw-env>
   <import>
     <file>/offcodes/checksum.odf</file>
     <bindname>Hydra.net.utils.Checksum</bindname>
     <reference type="Pull" pri="0"></reference>
     <GUID>6060843</GUID>
   </import>
 </sw-env>
```

Figure 4.6: ODF - Part II

- ***Link***: The Link constraint is denoted as $\alpha \overset{Link}{\Leftrightarrow} \beta$. This is the default constraint from α to β, which actually poses no constraints: α and β may or may not be mutually offloaded (to the same or different target device). It does, however, indicate that at least one of the Offcodes needs the other to function.
- ***Pull***: The Pull constraint is denoted as $\alpha \overset{Pull}{\Leftrightarrow} \beta$. This reference is used to ensure that both Offcodes will be offloaded to the **same** target device.
- ***Gang***: The gang constraint is denoted as $\alpha \overset{Gang}{\Leftrightarrow} \beta$. This constraint is used to ensure that both Offcodes will be offloaded to **their respective** target devices. That is, if α is offloaded, β will be too, albeit on perhaps a different device.
- ***Asymmetric Gang***: This constraint is denoted as $\alpha \overset{\sim Gang}{\rightarrow} \beta$ and provides the asymmetric version of Gang. Offloading β doesn't implies offloading α.

Figure 4.7: Offcodes's Constraints

The second part of an ODF describes the Offcode's dependencies on peer Offcodes. This section enables a developer to "design" the offloading process that will occur at deployment time. HYDRA provides several constraints presented in Figure 4.7 that can be used between any two Offcodes denoted by α and β. The set of Offcodes and related constraints form an *Offloading Layout Graph*. The runtime (recursively) processes an Offcode's ODF file to produce such a graph which is later used by the runtime for deciding on the actual placement of Offcodes.

Note that there is no *Asymmetric Pull* constraints as the motivation for using *Pull* is a tight interaction between two Offcodes. Enabling asymmetry may result in the placement of two Offcodes in two different execution domains. Figure 4.6 presents the mechanism by which a constraints is set on an Offcode reference. In this example, a *Pull* constraint is set for the peer Offcode denoted by: "Hydra.net.utils.Checksum".

The last part of the ODF is concerned with device mappings. In order to enable dynamic mapping between Offcodes and peripheral devices, on different hosts configurations, a developer is required to supply a list of potential target *device classes* that can be used for offloading.

Figure 4.8 a sample Offcode for which the developer indicated the classes of potential devices on which it can operate. It is the runtime's responsibility to locate an instance of such an Offcode which is suitable for running at one of the local devices that is in one of the listed classes.

```
 <!- device classes ->
 <targets>
   <device-class id=0x0001>
     <type>NIC</type>
     <mac>ethernet</mac>
     <bus>pci</bus>   <!- (optional) ->
     <rate>1000</rate> <!- (Mbps) ->
     <vendor>3COM</vendor> <!- (optional) ->
   </device-class>

   <device-class id=0x0002>
     <type>NIC</type>
     <mac>myrinet</mac>
     <rate>10000</rate> <!- (Mbps) ->
   </device-class>
 </targets>
</offcode>
```

Figure 4.8: ODF - Part III

Alternatively, the developer can specify the exact target for each Offcode using an Offcode's URL.

4.4 Deployment Process

This section provides a description of the deployment process that is performed by the HYDRA runtime. Figure 4.9 presents the control flow of the deployment process.

Once an Offcode is created by calling the *CreateOffCode* API, the appropriate Offcode ODF files are recursively processed by the runtime to construct the application's offloading layout graph. Following that, the runtime determines the mapping between the Offcode device requirements and the physical devices that are installed at the specific host. For this purpose, HYDRA uses the runtime resource management module to obtain a list of local HYDRA capable devices (e.g. devices that execute the HYDRA runtime). Finally, an Offcode instance (object file) must be selected for each given Offcode. Typically, the runtime uses a local library that is used for storing the actual instances of the Offcodes. The library has a simple hierarchical structure as presented in Figure 4.10. The library tree is first sorted by the device class and further by class specific properties. For example, network interface cards (NICs) are further categorized according to their Media Access

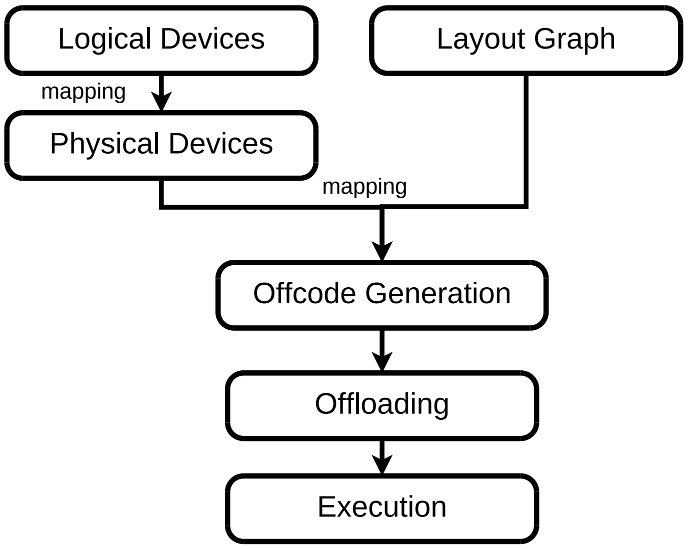


Figure 4.9: Deployment Control Flow

Control protocols (MAC), thus the *mac* identifier (also depicted in the Offcode's ODF file) is also used for traversing the tree.

```
- lib
    |
    - offcodes
            |
            - NIC
                |
                |- ethernet
                    |      |- GUID1
                    |           | - 3com
                    |               |- socket.oc
                    |               |- socket.odf
                    |           |- marvell
                    |               |- mrvl_socket.oc
                    |      |- GUID2
                    |- myrinet
                    |      |- GUID2
                    |           | - myricom
            - GPU
                ...
            - DISK
                ...
```

Figure 4.10: Offcodes Library Structure

As an example, assume that an instance for the "Hydra.net.utils.Socket" Offcode is to be located. First, the Offcode's GUID, the device class and the *mac* identifiers are extracted from the Offcode's ODF (see Figures: 4.5-4.8). In this case the tuples: *{NIC,ethernet,*$GUID = 7070714$*}* and *{NIC,myrinet,* $GUID = 7070714$*}* are read. Next, the resource management module is consulted to identify the currently installed devices (NICs) and to exclude irrelevant devices. E.g. if the local host has only ethernet connectivity then the myrinet device is excluded at this stage. Assuming this is the case, we are left with the first tuple extended with a list of locally installed NICs. Next, the tree is traversed according to this information and an Offcode instance is located. If such a mapping can not be allocated (due to resource limitations or incompatibility) the runtime tries to find an Offcode that is capable of executing at the host CPU.

The next step involves adapting the specific Offcodes instances to the target devices either by executing a corresponding compiler (for open source Offcodes) or by invoking the dynamic linkage process. The last phase is the actual offloading of the Offcode which is further described in Section 5.6 and discussed in details in [28].

Notice that the offloading layout is usually statically defined or set during deployment. The reasoning behind this is to minimize the overhead concerned with the offloading operations. The overhead imposed by enabling migration of Offcodes between devices is superfluous if this feature is rarely used.

Chapter 5

Architecture

In this chapter we present the design of the runtime system. The system implements the model and provides facilities for programming, testing, deploying, and managing OA-applications and Offcodes. Both the host OS and the target device firmware must support the interfaces defined by the programming API and implement the runtime functionality. A critical decision is to modularize the framework into independent parts, so that modifying one will not affect the rest.

The bottom half of the runtime system comprised of library requirements for a particular target device. Such libraries may be provided by the device manufacturer, system integrator, or by researchers and the open source community. The upper half of the runtime system exists on the host as operating system extensions. Our host implementation for Linux is modular, in that it maintains strict separation between device-specific code and generic code. It is implemented as a set of kernel modules that are loadable on demand and do not require kernel source code modifications.

5.1 HYDRA Components

The HYDRA runtime is comprised of several components as shown in Figure 5.1. It is accessed through an offloading access layer that consists of a user-level library linked to each OA-Application, and a kernel-level set of generic services.

The kernel layer consists of several functional blocks. The *System Call Management* and *Offloading API* blocks implement the various APIs defined in the programming model. The *Channel*

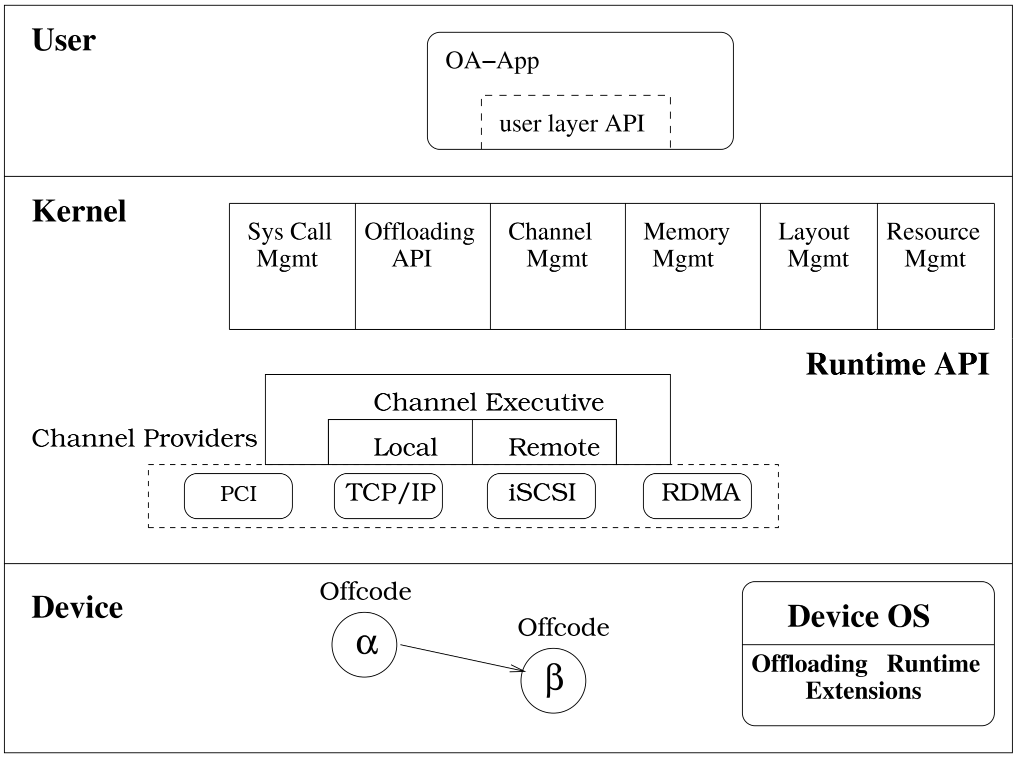


Figure 5.1: System Architecture

Management unit manages the channels by interacting with the *Channel Executive*. This module handles channel creation by using a particular *Channel Provider*. These providers are target-specific and provided as part of the driver for each programmable device. A channel provider creates various specialized channel types to the device and provides a cost metric regarding the "price" for communicating with the device through a specific channel, in terms of latency and throughput. The executive uses this capability information to decide on the best provider for a specific Offcode. The *Resource Management* unit keeps track of all active Offcodes and related resources. Resources are managed hierarchically to allow for robust clean-up of child resources in the case of a failing parent object. The *Memory Management* module exports memory services such as user memory pinning that is used by zero-copy channels. The *Layout Management* unit performs layout related functionalities such as analyzing the offloading layout graph. This unit receives the offloading layout graph as input and produces the mapping between Offcodes and target devices. The module can be easily extended to support future offloading constraints.

5.2 Offcode Internals

Offcodes are the building blocks of OA-applications. Each Offcode has state (data members), behavior (operations on data) and a thread of control that is initialized by the HYDRA runtime. Offcodes define and implement interfaces which are globally identified. An Offcode interface is designed as a function table (much like a C++ abstract class) that is accessed through a virtual table pointer. A virtual table enables better separation (encapsulation) between the interface and the implementation, it can be used by several instances of the same Offcode thus decreasing the required memory and achieves a binary level standard.

```
/* A pointer to an offcode interface */
typedef struct IOffcode *PIOFFCODE;

/**
 * the offcode’s basic virtual table
 * real offcodes extend this interface.
 */
struct IOffcodeVtbl {
    ErrorCode (*QueryInterface)(PIOFFCODE pThis,
                                REFIID iid,
                                void** ppObject);
    UINT32    (*AddRef)        (PIOFFCODE pThis);
    UINT32    (*Release)       (PIOFFCODE pThis);
    ErrorCode (*Initialize)    (PIOFFCODE pThis);
    ErrorCode (*StartOffcode)  (PIOFFCODE pThis);
    ErrorCode (*StopOffcode)   (PIOFFCODE pThis);
    ErrorCode (*GetOOBChannel) (PIOFFCODE pThis);
        ...
};
```

Figure 5.2: An Offcode Virtual Table

Figure 5.2 and Figure 5.3 present the related Offcode data structure. The Offcode’s virtual table (Figure 5.2) contains a set of function pointers that provide the basic Offcode’s functionality. For example, every Offcode must implement the *Initialize* and *StartOffcode* methods that are called at deployment time. The Offcode interface, denoted by *IOffcode*, is merely a data structure containing the Offcode’s virtual table. Offcodes should extend the basic *IOffcode* interface with specific Offcode’s functionalities.

```
/**
 * An offcode interface contains a pointer to an offcode vtable
 * which hold the offcode’s functionalities.
 */
typedef struct IOffcode {
    const struct IOffcodeVtbl *v;
} IOffcode;

/** declares an offcode interface */
#define DECLARE_OFFCODE_INTERFACE(iname)   \
    typedef struct iname { \
        const struct iname##Vtbl *v; \
    } iname;
```

Figure 5.3: An Offcode Interface

For example, Figure 5.4 and Figure 5.5 present the declaration and implementation of the *Heap* Offcode that resides in the *Hydra.Runtime* package. Upon initialization, the HYDRA runtime creates an *IHeap* data structure and sets its virtual table pointer to reference the *IHeapVtbl* structure (denoted as “heapMethods” in Figure 5.5). The Offcode is then registered at the runtime which creates a mapping between the Offcode’s URL and the *IHeap* data structure.

```
typedef struct IHeap *PIHEAP;
typedef struct IHeapVtbl {
    /* default offcode interface (IOffcode) */
    DECLARE_DEFAULT_OFFCODE(PIHEAP);
    /* Heap specific interface */
    void* (*Alloc)(PIHEAP pThis, UINT32 size, UINT32 flags );
    void  (*Free)(PIHEAP pThis,const void* pMemory);
} IHeapVtbl;

/* now define the offcode’s interface */
DECLARE_OFFCODE_INTERFACE(IHeap);
```

Figure 5.4: A Heap Virtual Table

```
/* heap specific methods implementation at our nic (nicos) */
void* HeapAlloc(PIHEAP pThis, UINT32 size, UINT32 flags )
{
    return (void*)nicos_malloc(size);
}

void HeapFree(PIHEAP pThis,const void* pMemory)
{
    nicos_free((void*)pMemory);
}

IHeapVtbl heapMethods = {..., HeapAlloc,HeapFree};
```

Figure 5.5: A Heap Offcode

5.3 Call Internals

A *Call* object is a data structure that encapsulates the relevant information needed to invoke an Offcode's method (see Figure 5.6). The structure is shared between the host and target devices.

```
typedef struct call_t
{
    u16 type;
    u16 status;
    u16 code;
    u16 target_id;

    /* the memory input descriptor */
    struct mem_desc_t in;

    /* the memory output descriptor */
    struct mem_desc_t out;

    /* an opaque data for internal use */
    u64 cookie;
} __attribute__ ((packed))  call_t;
```

Figure 5.6: A Call Object

A *Call*'s type is either *IN_CALL* or *INOUT_CALL*. The former corresponds to methods that do not return data, thus only contain an input buffer, while the latter corresponds to methods that do return data, thus contain an input *and* output buffers. *Call* buffers are described by memory descriptors that contain the buffers' addresses (64 bit) and length (Figure 5.7).

```
typedef struct mem_desc_t {
    u64 address;  /* buffer address */
    u32 len;      /* buffer size in bytes */
}  __attribute__ ((packed)) mem_descr_t ;
```

Figure 5.7: A Memory Descriptor

Upon creation, a call's *status* is set to *CALL_STATUS_PENDING*. The device can follow a simple or extended invocation scheme. In the simple invocation scheme, the device notifies the caller once the invocation has been completed. The extended invocation scheme consists of two acknowledgements phases. The first, occurs once the input buffer has been consumed by the device. The second acknowledgment occurs once the invocation is completed. The extended invocation scheme enables a developer to quickly release the input memory buffer when it is no longer needed by the device. Since zero-copy channels use DMA, the buffer can not be released voluntarily. Once the device DMAs the input buffer, it sets the call's status to *CALL_STATUS_ACKED*. When the call is completed (e.g., the data is ready at the output memory buffer), the call's status is changed to *CALL_STATUS_FINISHED* and the host is informed. The call's *code* field indicates the invocation call error code and the *target_id* denotes the target of the invocation (which is typically the target device's runtime).

```
/* call status */
#define CALL_STATUS_PENDING            (1«13)  /* 0x2000 */
#define CALL_STATUS_ACKED              (1«14)  /* 0x4000 */
#define CALL_STATUS_FINISHED           (1«15)  /* 0x8000 */
```

Figure 5.8: Call Status

A user typically creates a call object and sends it to the target device via a connected channel (see Section 5.5). The kernel holds an extended representation of the *call_t* object in the form of a *kcall_t*. This structure is used by the runtime for updating the user's calls status and for pinning or unpinning user buffers before or after DMA operations. For brevity, we omit the details of this data structure and corresponding operations.

5.4 Call Encoding

HYDRA provides a generic encoding scheme that enables a developer to choose a unique encoding per invocation. Typically, the call's input buffer contains a fixed length header with the encoding information. A developer can use a custom encoder or a standard one (see Figure 5.9).

```
/* encoding types */
#define ENC_CUSTOM              (1«0)
#define ENC_SOAP                (1«1)
#define ENC_XML_RPC             (1«2)
#define ENC_MAX_TYPE            (1«5)
```

Figure 5.9: Call Encoding

5.4.1 Custom Encoding

This section present a simple custom encoding scheme that is used in the HYDRA framework for efficient data transfer. Figure 5.10 presents the custom encoding header which holds the encoding information.

```
typedef struct CustomEncodingHeader {
  union  {
    struct {
      /* the target offcode URL */
      char  targetOffcodeBindName[OFFCODE_URL_LEN];
      /* method id to invoke (from the offcode interface)*/
      UINT8 method_id;
      /* method version */
      UINT8 method_ver;
      /* number of parameters */
      UINT8 param_count;
      /* params endianess (BIG_ENDIAN, LITTLE_ENDIAN)*/
      UINT8 endianess;
    } header;
    char data[32];
  } input;
} CustomEncodingHeader;
```

Figure 5.10: Custom Encoding Header

The header describes the invocation's target Offcode, the method identifier and version, and the number of parameters. Parameters are data structures (Figure 5.11) that contain information regarding their type and values (Figure 5.12).

```
typedef struct Param
{
    ParamType type;
    UINT8 data[0];
} __attribute__((__packed__)) Param;
```

Figure 5.11: Parameter Data Structure

The functionality required in order to create *Call* objects adhering to a custom encoding scheme is provided in the form of a *CustomEncoder* pseudo Offcode.

```
typedef enum ParamType {
    PARAM_RAW = 0,
    PARAM_CSTRING = 0x01000001,
    PARAM_UINT8   = 0x08000008,
    PARAM_UINT16  = 0x18000018,
    PARAM_UINT32  = 0x28000028,
    PARAM_UINT64  = 0x38000038,
    PARAM_SINT8   = 0x48000048,
    PARAM_SINT16  = 0x58000058,
    PARAM_SINT32  = 0x68000068,
    PARAM_SINT64  = 0x78000078,
    PARAM_BOOL    = 0x02000002
} ParamType;
```

Figure 5.12: Parameter Types

Figure 5.13 presents a sample usage of the interface for creating such a call. The call's target Offcode is "Hydra.utils.Tracer" and the method identifier is "0x07" which corresponds to the method "SayHello". This method takes a fixed length character array (*msg*) and an integer (*count*) which determines how many time the array should be traced out (at the target device). Once a reference to the custom encoder has been obtained (line 1), the call's parameters are initialized (lines 2-6). Then, the call object is created (lines 7-9) and the header is initialized (line 10). Once all of the parameters have been added to the call (lines 11-12), the call is ready to be passed on to the target device via a connected channel.

```
// get the runtime call encoder
IEncoder* pEncoder = GetCustomEncoder();
// create the call parameters, we will have 2 of them
Param* params[2];
const char* str = "Hello from user";
int c = 4;
params[0] = pEncoder->v->CreateMethodParam(pEncoder,PARAM_CSTRING,
                                       (void*)str,strlen(str)+1);
params[1] = pEncoder->v->CreateMethodParam(pEncoder,PARAM_UINT32,
                                       (void*)&c,sizeof(UINT32));
c = pEncoder->v->GetParamsSize(pEncoder,params,2,TRUE);
// create the call object
CallAttr attr = {TYPE_IN,ENC_CUSTOM, ENC_NONE, RUNTIME,
                 OFFCODE, c, 0 };
ICall* pMyCall = CreateCall(&attr);
// now setup the header
pEncoder->v->InitCallEncoding(pEncoder,pMyCall,0x07,0x00,
                             "Hydra.utils.Tracer");
// add the parameters to the call
pEncoder->v->AddMethodParam(pEncoder,pMyCall,params[0]);
pEncoder->v->AddMethodParam(pEncoder,pMyCall,params[1]);
```

Figure 5.13: Creating a Custom Encoded Call

5.4.2 SOAP Encoding

As SOAP is de facto the standard for web services invocation, we have chosen to provide a prototype implementation, for our programmable NIC, that will enable to encode a method invocation using SOAP. Our implementation is based on the the gSOAP web services toolkit [67]. This toolkit offers an easy to use XML to C/C++ language binding by using an extended C/C++ compiler. The gSOAP compiler generates efficient XML serializers for C and C++ data types that are used by the HYDRA runtime.

The basic gSoap functionality over TCP has been modified to use HYDRA APIs. For example, instead of invoking the regular TCP “connect” API, gSoap is set to invoke “CreateChannel”. Upon receiving a buffer on such channel, the NIC’s runtime invokes a handler which in turn retrieves the buffer and calls gSoap to parse it and execute the corresponding function. Once this is accomplished, and the function returns a value, gSoap serializes the return value into the call’s output buffer. Once the call is finished, the client side code uses gSoap to deserialize the buffer and return a value as if the function has been executed locally.

Figure 5.14 presents a sample user program that uses the gSoap framework in order to invoke a method implemented at some programmable NIC. Again, the invocation's target Offcode is "Hydra.utils.Tracer" and the method is "SayHello". The figure clearly shows that the gSoap framework simplifies the invocation process and removes the burden of manually constructing a call object.

```
// include generated proxy and SOAP support
#include "soapH.h"
#include "tracer.h"
int main() {
  char* msg = "Hello from user";
  int num = 5; int status = -1;
  struct soap soap;     // allocate gSOAP runtime environment
  soap_init(&soap);     // must initialize
  soap_call_nic__SayHello(&soap, "Hydra.utils.Tracer", "",
                          msg, num, &status);
}
```

Figure 5.14: Creating a SOAP Encoded Call

The method signature *soap_call_nic__SayHello* is automatically generated by the gSoap compiler invoked on the method declaration file presented in Figure 5.15. Developers should merely need to implement the method using the target device's runtime APIs.

```
//gsoap nic service name:       tracer
//gsoap nic service style:      rpc
//gsoap nic service encoding:   encoded
//gsoap nic service namespace:  urn:xmethods-delayed-quotes
//gsoap nic service location:   http://services.xmethods.net/soap
//gsoap nic service method-action: sayHello ""

int nic__SayHello(char *msg, int count, int* status);
```

Figure 5.15: Method Declaration (H file)

Figure 5.16 and Figure 5.17 present the XML format of a typical request and response associated with this specific method.

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope>...
 <SOAP-ENV:Body SOAP-ENV:encodingStyle=...>
  <nic:SayHello>
   <msg></msg>
   <count>0</count>
  </nic:SayHello>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 5.16: SOAP Request

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope>...
 <SOAP-ENV:Body SOAP-ENV:encodingStyle=...>
  <nic:SayHelloResponse>
   <status>0</status>
  </nic:SayHelloResponse>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 5.17: SOAP Response

5.5 Channel Internals

A channel is a transport abstraction used by Offcodes to communicate with each other and with OA-applications. Underlying every channel is a simple protocol capable of transferring data from one device to another. Typically, the device driver of the target hardware implements such a mechanism and exports this capability to the *Channel Executive* module. For example, a DMA master device exports a "zero-copy" channel capability thus enables the construction of zero-copy channels to the device.

Figure 5.18 shows a sample zero-copy channel architecture implemented for a programmable NIC. The right side of the figure presents the logical view as seen from the OA-application while the left side presents the internal architecture. The figure presents an OA-application that communicates with Offcode α through a proxy connected to a private channel identified by a channel descriptor. The HYDRA runtime maps each channel descriptor to an internal channel object that is created by the target device channel provider. This specific provider constructs two kernel buffer

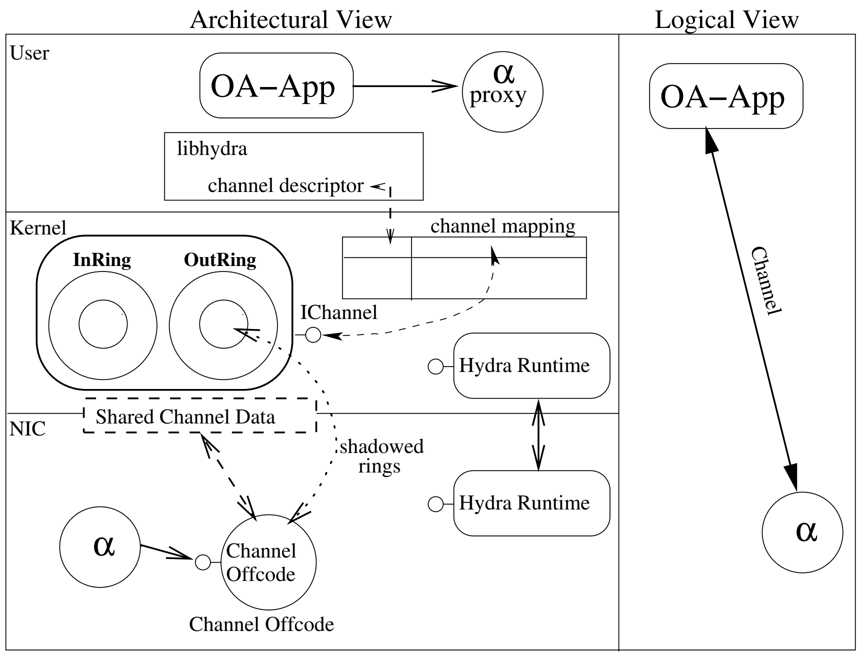


Figure 5.18: Example Zero-Copy Channel

rings to communicate with the target Offcode. The *InRing* holds memory descriptors that point to host memory locations that contain the *Call* objects. Although a *Call* object usually contains a return descriptor for delivering the invocation return value, the *OutRing* is necessary since it contains pre-posted application descriptors that are used by the runtime at the device for spontaneous messages triggered by the Offcode. The channel endpoint at the device holds a shadowed copy of the ring descriptors; and, channel management is maintained using a dedicated shared memory region per channel. The *Call* object is copied using the NIC's DMA bus master capabilities to an internal buffer owned by the target Offcode. The *Call* is de-serialized and the Offcode is invoked. The Offcode uses the embedded return descriptor to DMA the return value back to the application and optionally notifies the application using an event (usually interrupt) described by the shared memory region.

The described invocation process uses a channel provider that invokes a device specific method called: *invoke*. For example, the *invoke* method for our programmable NIC is presented in Figure 5.19. A typical invocation starts by mapping the call's virtual address to a bus address (lines: 10-12), registering the call in a user specific data structure (line 14) and finally, manipulating the device's registers which triggers an event at the device (line 17).

```
int invoke(struct ace_private *ap,
           kcall_t* kcall, user_context* uc)
{
    struct cmd issueCmd;
    int err=0;
    u64 dma_addr = 0;
    u32 pars[4];
    cmd_params_t* pParams = (cmd_params_t*)pars;

    issueCmd.evt = TG_METHOD_INVOKE;
    issueCmd.code = 0;
    issueCmd.idx = 0;

    dma_addr = pci_map_page(ap->pdev, virt_to_page(kcall),
                            offset_in_page(kcall),
                            sizeof(*kcall),PCI_DMA_TODEVICE);
    pParams->len=3;
    set_hostaddr(((tg_hostaddr_t*)pParams->params),dma_addr);
    pParams->params[2]=kcall->call.target_id;
    err = ht_insert(kcall, kcall, uc->calls);
    if (unlikely(err < 0))
            goto reg_err;
    if (ace_issue_cmd_sleepy(ap->regs, &issueCmd, pParams)<0)
        goto issue_err;
    return 0;
}
```

Figure 5.19: Sample Invoke Method

5.6 Offcode Dynamic Loading

Supporting dynamic Offcode loading is an important building block in the HYDRA framework. We have considered different approaches for implementing dynamic loading. The simple solution would be to hand over the Offcode to the target device and require that each device implement a simple Offcode loader. However this naive solution is quite expensive in terms of device resources. Another approach would be to fully perform the linking process at the host, and only transfer the Offcode when it is ready to be deployed (at a specific memory region). The device's loader will merely need to initialize the Offcode and execute it.

HYDRA runtime is built to support both approaches. HYDRA support for dynamic offloading is provided by a set of device-specific loaders that implement a generic interface for Offcode loading.

The interface is intended to be implemented by the device driver of each target peripheral. Each loader can decide whether to transfer the Offcode as is, or to perform some processing at the host first, depending on features of the target.

As a proof of concept, we have created such a loader for our programmable network card [28]. The dynamic offloading logic is implemented both in the device and in the host. A device-specific host-based loader is implemented at the NIC's driver; it uses the OOB-channel of the device's runtime to communicate with the target device loader, which is actually a pseudo Offcode at the target device (identified by the Offcode URL: "Hydra.Runtime.Loader"). Figure 5.20 presents the message transfers that occur in loading a single Offcode.

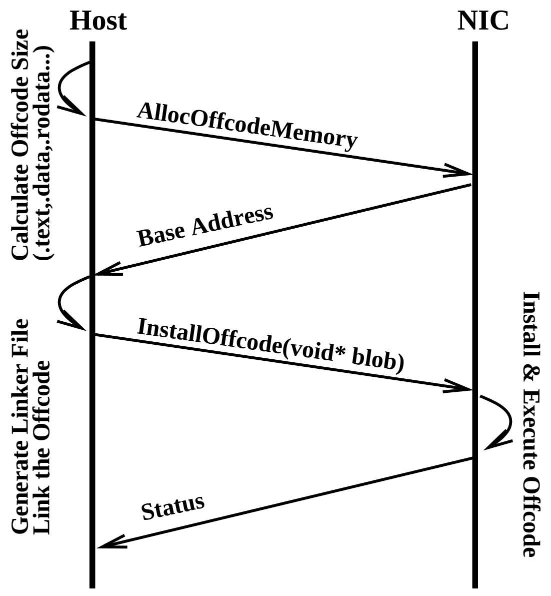


Figure 5.20: Offcode Dynamic Loading Flow

Once the host-based loader calculates the Offcode's size, it invokes the *AllocateOffCodeMemory* exported by the device's loader. This method allocates the memory region that will be used to store the Offcode binary and returns the device's memory address to the caller. The host-based loader dynamically generates a linker file adjusted by the returned address and links the Offcode object. It then transfers the linked Offcode to the target device where it is placed and executed. All the above interactions make use of the OOB channels that are created for the host and target HYDRA runtimes.

Chapter 6

NICOS Case Study

This chapter presents a Network Interface Card Operating System called:NICOS. The motivation behind NICOS is to facilitate the evaluation of the HYDRA framework. As a prototype device's OS, the generic HYDRA framework has been fully integrated within NICOS. All HYDRA's sample applications have been built on top of NICOS (see Chapter 8).

6.1 NICOS Environment

NICOS target device is a programmable NIC based on the Tigon2 chipset. The Tigon programmable ethernet controller is used in a family of 3Com's Gigabit NICs. Figure 6.1 presents the NIC's architecture. The NIC supports a PCI host interface and a full-duplex Gigabit ethernet interface. The Tigon has two 88 MHz MIPS R4000-based processors which share access to external SRAM. Each processor has a one-line (64-byte) instruction cache and a private on-chip scratch pad memory, which serves as a low-latency software-managed cache. Hardware DMA and MAC controllers enable the firmware to transfer data to and from the system's main memory and the network, respectively.

The Tigon controller uses an *event-loop* approach instead of an interrupt driven logic. The motivation is to increase the NIC's runtime performance by reducing the overhead imposed by interrupting the host's CPU each time a packet arrives or a DMA request is ready. Furthermore, on a single processor the need for synchronization and its associated overhead is eliminated.

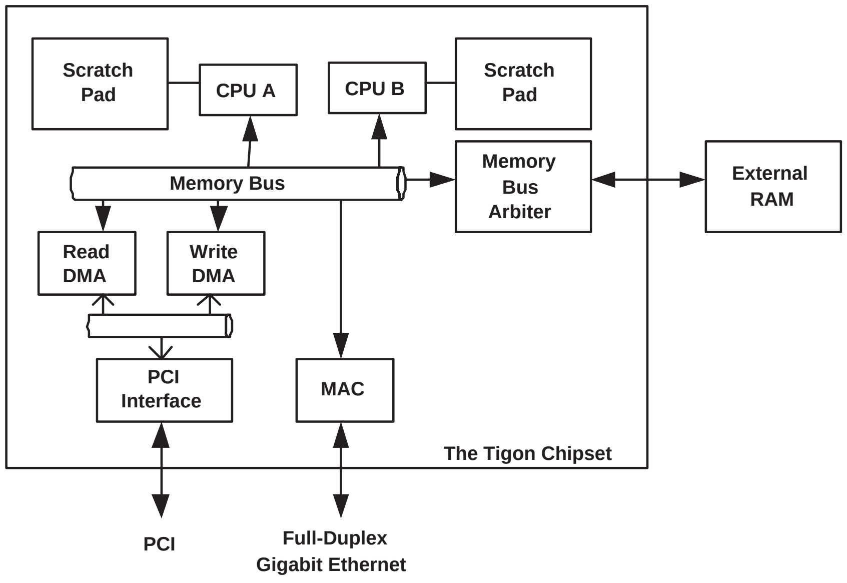


Figure 6.1: Tigon Controller Block Diagram

6.2 NICOS Services

This section presents NICOS services starting with the memory management service, NICOS task management and NICOS networking and filtering APIs. Following that a detailed description of the NICOS scheduling framework is provided.

6.2.1 Memory Management

NICOS has to allocate memory each time a task, a queue or a packet is created. NICOS default memory allocation algorithm is based on the "boundary tag method" described in [78], which is suitable for most applications. Implementing a "generic" memory allocation mechanism is problematic. Since different realtime systems may have very different memory management requirements, a single memory allocation algorithm probably will not be appropriate. To get around this problem the memory allocation APIs provided in NICOS can be easily replaced by using the filtering APIs (see Section 6.2.4). A user's task can easily replace the default methods by installing a special kind of a filter. The registered method (i.e., the "filter" action) will be called instead of the default allocation routine. NICOS memory allocation APIs can also enable a developer to choose the *target* of the allocated memory. Memory consuming applications can allocate memory at the host. The memory is transparently accessed using DMA. This scheme is also suitable for developing OS bypass protocols, which removes the kernel from the critical path and hence reduces the end-to-end latency.

6.2.2 Task Management

NICOS provides several task management APIs that enable a developer to create/destroy tasks and to control their lifecycle state. The API enables a developer to create a periodic or non-periodic task, to yield, sleep, suspend, resume and kill a task. Although periodic tasks can be implemented by a developer on top of a sleep API, an explicit facility for periodic tasks has been added so the OS is fully aware of them. Such a design allows the OS to minimize the ready-to-running[1] latency. Providing the timeliness guarantees required by NICOS has been a major challenge due to the non-preemptive architecture of these NICs.

6.2.3 Networking

The current networking API is very simple. NICOS provides only a single method that sends raw data. The data is provided by the developer and includes all of the necessary protocol headers. NICOS supports synchronous and asynchronous send calls. The asynchronous ones are non-blocking. When using the synchronous mode, the execution is blocked until frame transmission is completed. Upon completion, the provided callback is called. Receiving a packet is currently done only via filter registration.

6.2.4 Filtering

When deciding which functionality is needed to be offloaded to the NIC, one should look for common building blocks in today's networking applications. The ability to inspect packets and to classify them according to specific header fields is such a building block. For instance, the classification capability is useful for firewall applications, applying QoS for certain traffic classes, statistics gathering, etc. NICOS services include a packet filtering and classification capabilities.

[1]The time from the moment a task becomes ready-to-run until it starts execution.

In NICOS, a filter is a first class object. As such, it can be introspected, modified and created at runtime.

A sample code for installing a filter is presented in Figure 6.2. The "registerICMPFilter" method, registers a filter that drops all ICMP packets. Installing a filter is performed in two steps. In the first step, a pattern filter structure must be initialized (lines 1-7). This structure contain patterns that should be matched against each packet. In the second step, a filter is created (lines 8-9) and installed (lines 10-11) at either the receive path ("Rx filters"), the transmit path ("Tx filters") or both. Note that two kinds of filters exist: A *static filter* and a *dynamic filter*. The former matches a packet against a fixed pattern while the latter uses a custom callback function that is invoked for each received or transmitted packet.

```
    void registerPingDropFilters(void) {
    /* we would like to match ICMP packets */
1.  valueMask[0] = ICMP_PROTOCOL;
2.  bitMask[0] = 0x1; // match 1 byte
    /* start matching at ICMP_PROTOCOL_BYTE */
3.  pattern_filter.startIndex =ICMP_PROTOCOL_BYTE;
4.  pattern_filter.length = 1;
5.  pattern_filter.bitMask = bitMask;
6.  pattern_filter.numValues = 1;
7.  pattern_filter.valueMask = &valueMask;
    /* create the filter, add to Rx/Tx flows */
8.  pingDropFilter.filter_type = STATIC_PATTERN_FILTER;
9.  pingDropFilter.pattern_filter = &pattern_filter;
10. nicosFilter_Add(&nicosTxFilters,&pingDropFilter,DROP,NULL,
                    GENERAL_PURPOSE_FILTERS_GROUP,&pingFilterTxId);
11. nicosFilter_Add(&nicosRxFilters,&pingDropFilter,DROP,NULL,
                    GENERAL_PURPOSE_FILTERS_GROUP,&pingFilterRxId);
    }
```

Figure 6.2: Installing "Ping Drop" Filters

6.2.5 Scheduling

As discussed in Section 6.1 most high-end NICs do not support preemption. Schedulers for such non-preemptive environments usually use an event-driven model. For example, the programmable NIC used for evaluating NICOS, provides a special hardware register whose bits indicate specific events. This event register is polled by a “dispatcher loop” that invokes the appropriate handler. Once the event handler runs to completion, the dispatcher loop resumes.

Using a common real time scheduling algorithm for such devices yields a great inefficiency in the resulting schedule. NICOS scheduling scheme, henceforth called: <Sched>++, is capable of extending any given non-preemptive scheduling algorithm with the ability to create finer-grained schedules. The scheme has been used for implementing an enhanced version of the *Earliest Deadline First (EDF)* scheduler.

6.3 <Sched>++ Algorithm

6.3.1 Common Schedulers

Several scheduling algorithms have been implemented for NICOS. The *Cyclic-Executive* scheduler [14] is the simplest one. The cyclic executive approach has several advantages: it is simple to understand, easy to implement and very efficient. Unfortunately, the deterministic nature of the algorithm requires careful design and massive testing in order to produce deterministic timelines.

A more flexible scheduling algorithm is the non-preemptive version of the EDF algorithm [41]. In EDF, the task with the earliest deadline is chosen for execution. In the non-preemptive version, the task runs to completion.

Both EDF and cyclic executive are not optimal for a non-preemptive environment. For a set of scheduable tasks, the resulting task schedule meets the tasks’ realtime requirements, however with a rather low CPU utilization. The following sections present the <Sched>++ algorithm. The algorithm utilizes the **compiler**’s capabilities in order to create an optimized tasks schedule.

6.3.2 Related Definitions

A task is a sequence of operations to be scheduled by a scheduler. A task system $T = \{T_1, \cdots, T_n\}$, where each task T_i is released periodically, is called a *periodic task system*. Each task T_i is defined by a tuple (e_i, d_i, p_i, s_i), where e_i is the task’s Worst Case Execution Time (WCET), s_i is the first time at which the task is ready to run (also known as the start time), d_i is the deadline to complete the tasks once it is ready to run, and p_i is the interval between two successive releases of the task. Thus, a task T_i is first released at s_i and periodically it is released every p_i. After each periodic release, at some time t, the task should be allocated e_i time units before deadline $t + d_i$. A *non-periodic* task is a task that is released occasionally, and at each invocation that task may require a different execution time. A *hybrid task system* is a system that contains both periodic and non-periodic tasks. To differentiate between the periodic and non-periodic tasks, a periodic task will be denoted as $\tilde{T}$.

<Sched>++ assumes a *hybrid task system*, where for each periodic task, $d_i = p_i$. To represent the runtime instance of a task, the notion of a *ticket* of a task is introduced. A ticket of a periodic task, $\tilde{T}_i$ is defined as the tuple (e_i, p_i, Pr_i), where e_i and p_i are the execution and the period of the task, and Pr_i is the task's priority. The ticket of a non-periodic task, T_j, is (e_j, Pr_j). This assumes that any type of task scheduler used by the OS can be extended using this ticket.

6.3.3 Algorithm Overview

<Sched>++ uses several compile-time techniques, which provide valuable information that can be used at runtime. The developer uses <Sched>++ specific compiler directives in order to define the system's tasks and tickets. The compiler uses these tickets as simple data structures in which it can store the calculated WCETs.

The compiler uses the generated control flow graph in order to calculate the WCET of the periodic and non-periodic tasks (and warns if the code should be annotated due to recursions, unbounded loops etc.). Typical periodic tasks are comprised of a single calculated WCET, while non-periodic tasks may be comprised of a set of WCETs. In this context, a WCET is defined as the worst case execution time between two successive yields.

The ability of a compiler to modify the developer's code, at predefined places, is also utilized. By modifying the code, the ticket primitive is maintained automatically. The enhanced compiler updates the ticket with the task's next WCET prior to each *yield* invocation. This technique also eliminates the need to introduce a complicated runtime structure that contains all the WCETs of a given non-periodic task. A *single* ticket is recycled to represent the next task segment WCET at runtime.

6.3.4 <EDF>++ Algorithm

In order to implement the enhanced version of the EDF algorithm, the ticket of a periodic task $\tilde{T}$ is extended to be (e, p, Nr, Nd, Pr), where the additional fields Nr and Nd are the next release time and deadline of the task, respectively. Figure 6.3 presents the main logic behind the <EDF>++ algorithm, which is invoked by the *Yield()* function call. Part I of the algorithm starts with the classical EDF algorithm. The algorithm selects the next periodic task T_{next} that has the earliest deadline among all periodic tasks that are ready to run.

```
       Yield() called from task $T_k$:

I:     $T_{next} = \{\tilde{T}_i | \tilde{T}_i.Nd = \min(\tilde{T}_j.Nd | \tilde{T}_j.Nr \geq t)\}$;

       /* If no periodic task is ready, then
       choose from the non-periodic tasks */
II:     if ($T_{next} = NULL$)
            $SlackTime =$ duration until next
                      periodic task is ready;
            /* Pick the next non-periodic task
            that will run at most 'SlackTime'
            time units */
            $T_{next} = PickNonPeriodicTask(SlackTime)$;

       /* if no task is ready, the Idle task
       will run for the time duration until
       the next periodic task is ready */
III:    if ($T_{next} = NULL$)
            $T_{next} = Idle\_Task(Timeout)$

       SwitchTo($T_{next}$);
```

Figure 6.3: <EDF>++ Scheduler

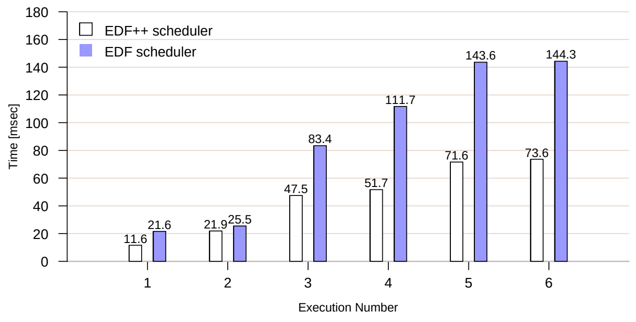
Part II of the algorithm is invoked when no periodic task is ready to run. The algorithm uses the tickets of the non-periodic tasks in order to select the next task to run. The chosen task should be able to run without jeopardizing the deadline of the next (earliest) periodic task. The scheduler considers the subset of non-periodical tasks that are ready to run, such that their next execution time is smaller than the slack time (the time until the next periodic task is ready). Among such tasks, the algorithm can use various criteria to pick the next task to be scheduled. For instance, one can use the algorithm in [59], which chooses a set of tasks that minimizes the remaining slack time. Any such algorithm would use the next execution time (WCET) of the tasks listed in their tickets.

When there is no suitable task for execution, the IDLE task is invoked until the next periodic task is ready to run (part III).

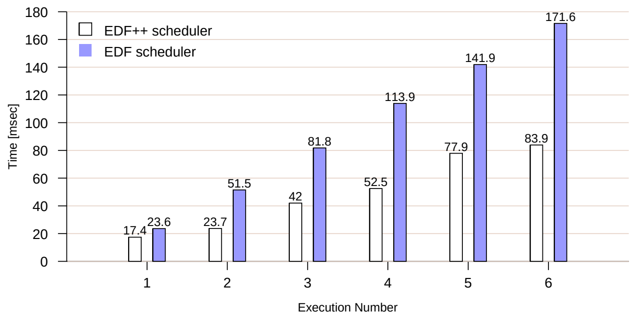
Notice that the scheduling algorithm attempts to schedule non-periodic tasks whenever there is an available time slot in the schedule. Available time slots may exist between periodic slots or whenever a task completes its execution ahead of time, which can only be determined at runtime.

6.3.5 <EDF>++ Evaluation

An experimental system with both EDF and <EDF>++ schedulers has been implemented. The task set includes twenty tasks where half of them are periodic. The system has been executed with various periods and constraints. On average, for plain EDF, the IDLE task has been executed 28.6% of the time, yielding a CPU utilization of 71.4%. For the <EDF>++ algorithm, the IDLE task ran 14.7% of the time corresponding to 85.2% CPU utilization, an increase of 20% in the system's throughput.



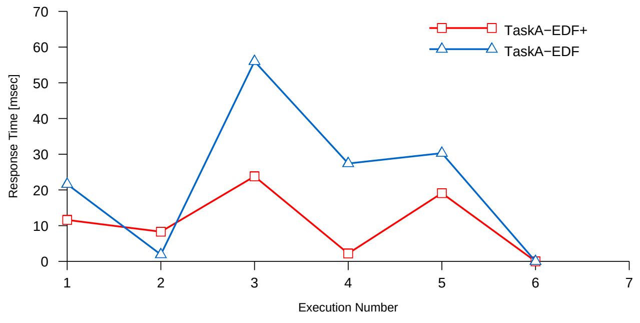
(a) Task A



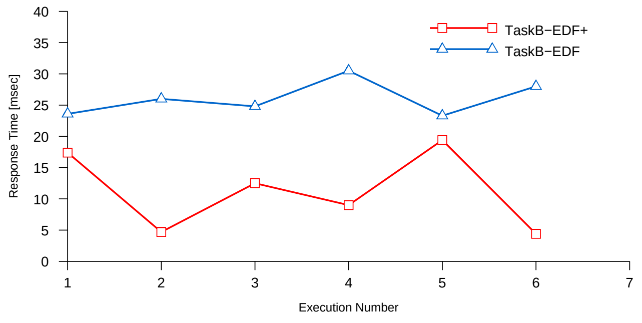
(b) Task B

Figure 6.4: Invocation Times

Figure 6.4 shows a sequence of invocation times for two sample tasks measured from the system's start time. The x-axis shows the number of invocations, where the y-axis presents the time when the specific invocation occurred. The response times, in-between invocations, for the non-periodic tasks are presented in Figure 6.5. For example, the average response time for task A, using <EDF>++, is $10.83ms$ with standard deviation of $8.51ms$ versus $22.86ms$ and $18.87ms$ using EDF (a 53% decrease in the average waiting time). For task B the values are: $11.23ms$ and $5.78ms$ against $26.03ms$ and $2.54ms$ (57% decrease in the average waiting time). The graphs clearly show



(a) Task A



(b) Task B

Figure 6.5: Response Times

that the response times for the non-periodic tasks using the <EDF>++ scheduler are improved. Regarding the response times for periodic tasks, the average response time is approximately the same ($1.37ms$ vs. $1.33ms$ with standard deviation of $2.49ms$ vs $2.39ms$). Thus, the improved response for the non-periodic tasks didn't affect the response time for the periodic tasks.

Chapter 7

Multi-User Environments

The strength of the proposed programming model lies in the the ability to reuse the Offcode components. On the one hand, reusability may simplify and speedup the development cycle, but on the other hand, in multi-user environments, reusing the same Offcode in several applications may substantially complicate the offloading layout design. Intuitively, the problem of defining an optimal offloading layout graph for a group of offload-aware applications may introduce an infeasible combinatorial problem. This section provides an Integer Linear Programming methodology (ILP) for optimizing such complex layouts. The purpose of such a formulation is to enable expression of every offloading layout graph as a set of linear equations. Any ILP solver can then be used to solve the equations given a target optimization function.

We provide the mathematical presentation of an offloading layout graph and later present, as an example, two possible criteria that could be used as target optimization functions. A detailed example of formulating a sample offloading graph is presented in [47]. As the simple graphs are trivially easy to solve, the strength of such a formulation is only apparent at complicated scenarios. Such scenarios make the offloading layout design process significantly more difficult. In such cases, a greedy solution does not provide an optimal solution, hence the need for such a formulation is apparent.

7.1 Formulation

As the offloading layout design essentially produces a graph, it is desirable to mathematically express the dependencies among the graph vertices (i.e., Offcodes). This section provides the ILP formulation that is required for optimizing the offloading layout graph.

7.1.1 Definitions

We begin by defining the basic elements of the layout graph. The layout graph $G = (V, E)$ includes the set of Offcodes as vertices, and the channel constraints among them are the edges. At deployment time the runtime associates with each node n (Offcode) a compatibility target vector $\vec{C_n}$ representing the potential target devices that can host the Offcode. Note that the host CPUs are included in the list of devices. Let $N = |V|$ be the total number of Offcodes, and let $K = |\vec{C_n}|$ be the number of HYDRA compatible devices.

NOTATION 7.1.1 *Let $\vec{C_n}$ be a constant binary bit vector. $C_n^k = 1$ if Offcode n* can *be offloaded to device k.* $\forall n \in N, k \in K$*,* $C_n^k \in \{0, 1\}$.

To simplify the presentation we assume that the first entry in each vector $\vec{C_n}$ corresponds to the host CPUs.

NOTATION 7.1.2 *Let $\vec{X_n}$ be the ILP output vector. $X_n^k = 1$ if Offcode n* should *be offloaded to device k.* $\forall n \in N, k \in K$*,* $X_n^k \in \{0, 1\}$.

The following equation guarantees a unique placement of each Offcode (a single Offcode can be offloaded to a single device).

$$\sum_{n=1}^{N} \sum_{k=1}^{K} X_n^k \cdot C_n^k = 1 \ . \tag{7.1}$$

Additionally, an Offcode n **is not** offloaded (remains in the host CPU) if $X_n^0 = 1$.

7.1.2 Constraints Formulation

For each one of the channel constraints (see Section 4.3), an integer linear equation is defined.

NOTATION 7.1.3 *Let $E^n_m = (m, n)$ be an edge from Offcode m to n.*

The following equations formulate the various channel constraints.

Pull Constraint:

$$\forall E^n_m \in \textit{Symmetric Pull}, \forall k : X^k_n = X^k_m \ . \tag{7.2}$$

Gang Constraint:

$$\forall E^n_m \in \textit{Symmetric Gang} : \sum_{k=1}^{K} X^k_n = \sum_{k=1}^{K} X^k_m \ . \tag{7.3}$$

Asymmetric Gang Constraint:

$$\forall E^n_m \in \textit{Asymmetric Gang} : \sum_{k=1}^{K} X^k_n \leq \sum_{k=1}^{K} X^k_m \ . \tag{7.4}$$

These equations are sufficient to represent the joint offloading layout graph as a set of linear equations.

7.2 Optimization Objectives

We have identified several optimization functions, two of which presented below. The list is by no mean complete, additional objectives functions can be easily added to address various applications needs.

1. Maximized Offloading – The trivial objective is to offload as many Offcodes as possible. The motivation for such a goal is to minimize the CPU usage and memory contention at the host:
$$\max \left(\sum_{n=1}^{N} \sum_{k=1}^{K} X^k_n \right) \ .$$

2. Maximize Bus Usage – This objective aim is to fully utilize the bus interconnect bandwidth among devices. A “Price” value is assigned to each Offcode. This value represents the estimated *average* bus bandwidth that is required by the specific Offcode. The bigger the value, the more bandwidth is required by the Offcode. In addition, we define a capability matrix per host. This matrix describes the *maximal* bus bandwidth between every two peripheral de-

vices. This matrix is used for limiting the number of offloaded Offcodes as the ILP solution must be limited by the physical busses limitations.

Chapter 8

Framework Evaluation

The previous chapters described the HYDRA system, including its programming model and its internal design. In this section we demonstrate the use of HYDRA through several sample applications.

8.1 TiVoPC

This section present a case-study for developing the TiVoPC application using our proposed framework. We focus on showing how HYDRA simplifies the design and development of offload-aware applications.

8.1.1 TiVoPC Architecture

The system architecture of the TiVoPC application is presented in Figure 8.1. The figure presents a client-server architecture comprised of a *Video Server* and a *Video Client*.

The *Video Server*, presented on the left hand side of Figure 8.1, corresponds to the cable-TV broadcaster. Typically, Network Attached Storage (NAS) devices are used to store the massive amount of broadcast media (MPEG movies, radio channels etc.). In order to emulate such a broadcaster, we have implemented a software-based server that is executed on a standard PC. The server reads the media from a NAS device which is mounted as an NFS device, and streams the media to the client as a stream of UDP packets.

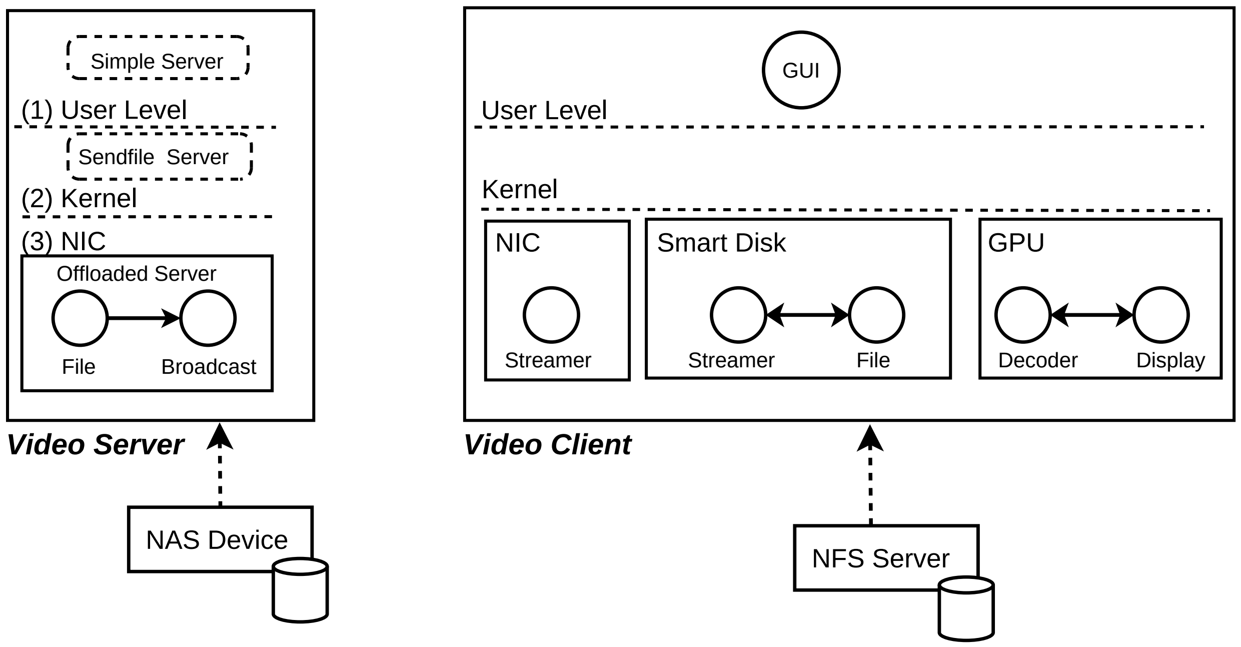


Figure 8.1: TiVoPC Software Architecture

The architecture of the *Video Client* is shown on the right hand side of Figure 8.1. The PC hosts the following programmable peripherals:

- **NIC** — This device is connected to the multimedia streaming server and intercept the transmitted UDP packets.
- **"Smart Disk"** — Although programmable disk controllers are common, in order to speed up prototyping, we have decided to emulate one by using a programmable NIC. Our "Smart NIC" exports a standard filesystem block device that interacts with an NFS server for storing the data (i.e., the streamed video is effectively stored on a remote disk). Essentially, we have created an NFS Offcode that implements various parts of the NFS-protocol.
- **GPU** — The graphics processing unit is responsible for rendering and displaying the movie on the screen.

8.1.2 TiVoPC Logic

As the programming model suggests (Section 4), the first phase in the development process should be designing the TiVoPC logic. This phase is usually performed without considering the physical placement of the various components. Figure 8.1 presents the following TiVoPC components.

- *GUI* — TiVoPC GUI contains a viewing area, for displaying the received video stream, and several controls used for rewinding, pausing and playing back the movie.
- *Streamer* — This component handles incoming packets. Specifically it should extract the network packet's payload that contains the three types of MPEG frames: the I-frame, P-frame and B-frame. The component should also process packets that are received from the storage device. The component implements a *callback* method which is invoked each time a packet is received. Upon invocation, the *Streamer* extracts the payload and passes it to the *Decoder* component.
- *Decoder* — This component is responsible for decoding the MPEG frame for later displaying it on screen. This component holds a reference to a *Display* component.
- *Display* — This component represents the display. For example, in a host level implementation this object could potentially wrap an OpenGL's FrameBuffer object or simply use a memory map of the GPU's physical memory (for the direct manipulation of the display).
- *File* — This component provides the basic file level APIs, such as open, read, write and close.
- *Broadcast* — This component is used at the *Video Server* for broadcasting the movie frames back to the client. This component provides unreliable message delivery as it uses UDP as its transmission protocol.

Some of the components could have been omitted. For example, the Streamer could directly access the local file system using the standard APIs, without the need for an additional *File* object. Alternatively, the Decoder could directly manipulate the display without the need for another level of indirection that is realized as the *Display* component. Although this observation is correct, introducing such objects improves the flexibility of the design. For instance, if a *Display* Offcode for the local GPU is found, either locally or in the vendor's Offcode library, it will be used at the GPU, thus increasing the overall application performance.

Once the components have been identified, we decide which of them should be implemented as Offcodes. Additionally, the Offcode communication channels should be also specified. Following are three characteristics that typically indicate a component should be implemented as an Offcode:

1. The component can use specialized capabilities that exist only at a peripheral device.
2. Offloading the component reduces the amount of traffic on host busses.
3. The component is tightly coupled to another Offcode.

In our example, all the components except for the GUI fall into one of these three categories and thus will be implemented as Offcodes.

8.1.3 TiVoPC Offloading Layout

The offloading layout of the TiVoPC application matches an Offcode to a peripheral device. The ODF discussed in Section 4.3 contains this information in addition to the Offcode's constraints regarding its peer Offcodes. For brevity we omit the ODF details and instead provide the considerations for designing the offloading layout as depicted in Figure 8.2.

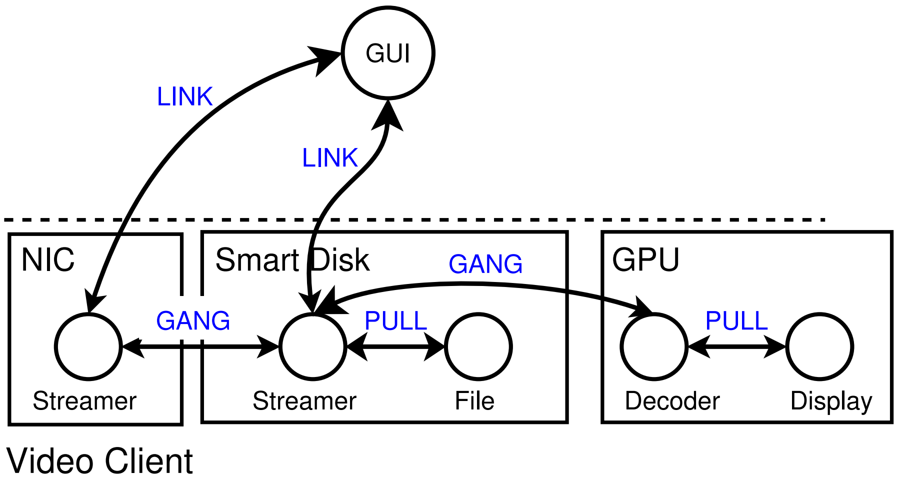


Figure 8.2: TiVoPC Offloading Layout

The *Streamer* Offcode resides at the NIC and at the "Smart Disk" devices. Reusing the same component at both devices is achieved by storing the received frames, without modification, at the storage device (so the source of the media packet becomes oblivious to this component). Since

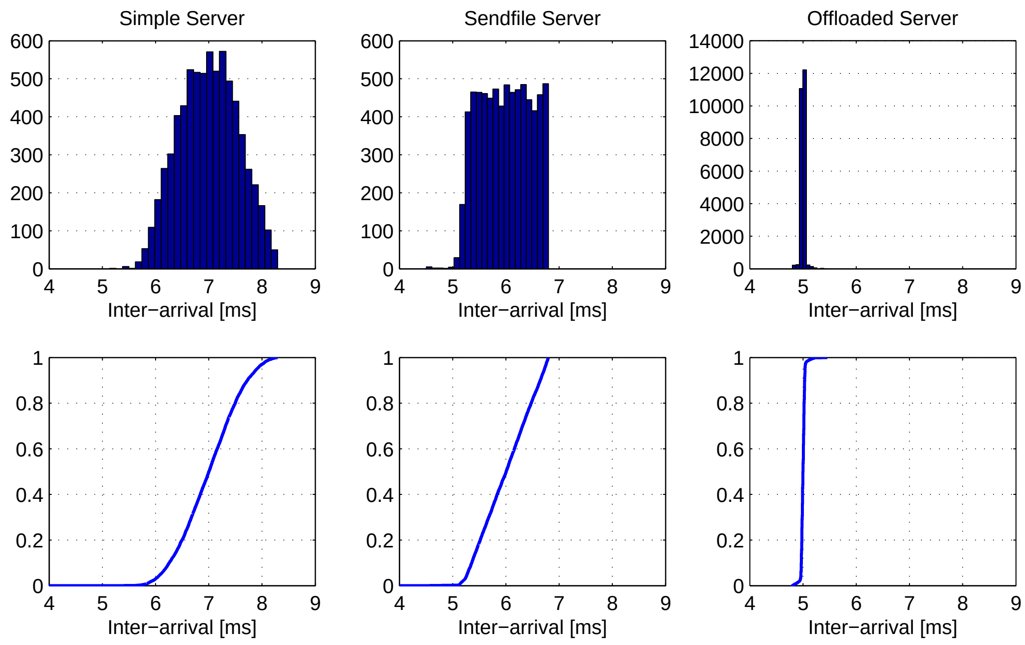


Figure 8.3: Jitter Distribution

we do not want packets to traverse the bus twice, a *Gang* constraint is imposed between the two components.

Intuitively, the *Display* Offcode should be placed at the GPU device, while the *Decoder* Offcode could be placed either at the NIC or at the GPU. In both cases, one bus transfer is required for transferring the media packet from the NIC to the GPU. The preference of placing the *Decoder* at the GPU comes from two reasons. First, the GPU may have specialized MPEG support on board. Second, a single *Decoder* could be used instead of duplicating the component at the NIC and at the "Smart Disk". In essence, requiring a *Gang* constraint between the two Offcodes will minimize the number of bus crossing operations. Therefore, the *Streamer* Offcode holds a *Gang* constraint to the *Decoder*, which holds a *Pull* constraint to the *Display*.

The *File* Offcode should reside at the "Smart Disk" and should be *Pulled* with the *Streamer* as both Offcodes tightly interact while the movie is stored/loaded from/to the storage device.

A simple *Link* constraint is sufficient between both *Streamers* and the *GUI* since the only information that traverse between them is control. As this is the default channel constraint, it can be omitted from the layout specification.

Once the application logic and the offloading layout have been coded, the communication channels between the various components are set. In the TiVoPC application, we have used a zero-copy read/write channel for all communication channels except for the two channels between the *GUI* and the *Streamer* Offcodes. Communication between the *GUI* and the *Streamers* utilize the default, low priority, *OOB-Channel*.

8.1.4 Benchmarks Description

Our experimental test-bed consists of two Intel Pentium IV computers, interconnected by a Dell PowerConnect 6024 Gigabit switch, having the characteristics listed in Table 8.1.

CPU speed	2.4GHz
RAM	512MB
L2 Cache	256KB
Cache line	64B
OS	Linux FC5, 2.6.15-1
NIC	3Com 3C985B-SX 1Gbps
GPU	nVidia 7800 GT
Inter-packet Interval	5 msec
Packet Size	1KB

Table 8.1: TiVoPC Application Test-bed

It should be noted that for demonstration purposes only, we did not send packets at video frames boundaries. What we did is to send the video stream in arbitrary chunks of 1KB, while maintaining the required bit rate. Specifically, for a video stream of 200KB/Sec we send 1KB chunks every 5ms. We executed the following benchmarks on an idle system.

Video Server Packet Jitter

Three versions of the *Video Server* have been implemented as indexed by the numbers 1–3 at the left hand side of Figure 8.1.

The first implementation (indexed by number 1) uses two UDP socket endpoints. Every 5 msec, a movie frame is read to a statically allocated buffer of size 1KB, then a connected UDP socket targeted at the client host is used for sending the packet to the TiVoPC client.

The second implementation (indexed by number 2) utilizes the “sendfile” system call. This call

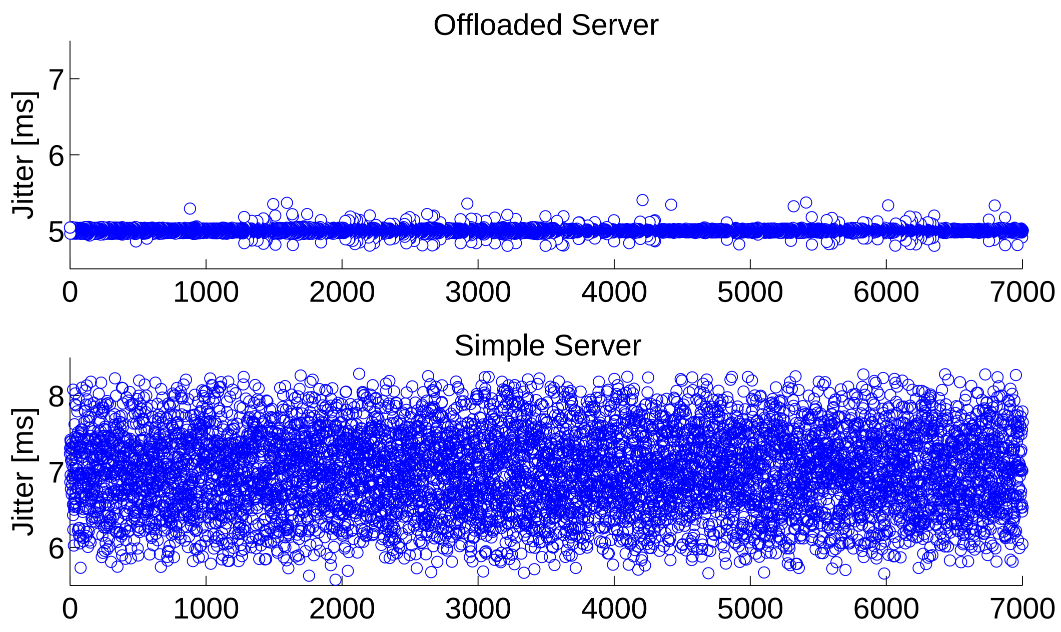


Figure 8.4: Inter-arrival times comparison

operates in two steps. In the first step, the file content is copied into a kernel buffer by the device's DMA engine. In our case, the server uses a NAS for storing the movies, hence the NIC is the one that acts as the DMA master. In the second step, a socket buffer is initialized with the required information about the location and length of the data just received. Scatter-gather hardware support is required at the networking device in order to be able to handle such a socket buffer. In cases where the hardware fails to support this feature, the CPU copies the data to the socket buffer.

The third implementation is an offload-aware server (indexed by number 3). This server is implemented as a simple Offcode residing at the networking device. It uses the *File* Offcode for reading the data in from the NAS device, and the *Broadcast* Offcode for transmitting the data back to the client.

Figure 8.3 shows, for each server implementation, a histogram and the corresponding cumulative distribution function (CDF) of packet jitter as measured at the client machine. A low level of jitter is more important than reliable delivery in video applications, as an unsteady packet rate is easily detectable by a human viewer.

Figure 8.3 clearly shows that the offloaded version of the streaming server produces a significantly lower jitter. This observation is further supported by its corresponding CDF. The user level version that uses “sendfile” produces better results than the “Simple Server” due to fewer context switches and data copying operations.

Figure 8.4 further presents a scatter plot of the inter-arrival times of the movie frames. The offloaded server is compared with the simple server. Table 8.2 provides the jitter statistics of received packets which corresponds to the execution of the three servers.

Scenario	**Median**	**Average**	**Std Dev**
Simple Server	6.99	7.00	0.5521
Sendfile Server	6.00	5.99	0.4720
Offloaded Server	5.00	5.00	0.0369

Table 8.2: Client Side Jitter Statistics

Scenario	**Median**	**Average**	**Std Dev**
Idle	2.90%	2.86%	0.09%
Simple Server	7.50%	7.50%	0.12%
Sendfile Server	5.90%	6.20%	0.08%
Offloaded Server	2.90%	2.86%	0.09%

Table 8.3: Server Side CPU Utilization

Video Server CPU Utilization

This benchmark is intended to validate our assumption that offloading certain parts of an application will reduce pressure on the host memory subsystem. The L2 cache miss rate that is experienced by the kernel is measured on the server during each one of the following tests. Samples were taken every 5 seconds during a 10 minute run. All measurements were normalized to the miss rate experienced by an otherwise idle system. Figure 8.5 shows the results.

Although the TiVoPC application is mostly I/O bound, executing it at the host incurs a 7% increase in the L2 cache miss rate.

The second implementation uses the “sendfile” API which avoids unnecessary buffer copies between kernel and user buffers. As indicated by Figure 8.5, the effect on the L2 cache is negligible. The reason becomes clear as the “sendfile” source code is examined. Since most of today’s network devices support scatter-gather operations, the kernel essentially follows a zero-copy data path between the two sockets. This approach reduces the number of context switches and totally eliminates data duplication inside the kernel.

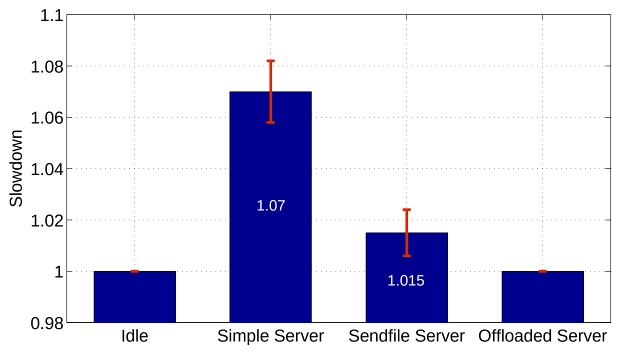


Figure 8.5: L2 Slowdown (Server Side)

Table 8.3 presents the CPU utilization at the server side. Each row corresponds to one of the three scenarios presented in Figure 8.5. Notice that the CPU utilization of the offloaded version of our server aligns with the *Idle* scenario results, as the host processor is unaware of the underlying activity.

Video Client Memory and CPU Utilization

The client side implementation, shown on the right of Figure 8.1, is more interesting from the offload point of view, as it involves five offloaded components and interesting constraints, compared to the two components in the server. But the overall performance results are more modest, thus we only give a brief overview to save space.

Scenario	**Median**	**Average**	**Std Dev**
Idle Client	2.90%	2.86%	0.09%
User-space Client	7.30%	6.90%	0.32%
Offloaded Client	2.90%	2.86%	0.09%

Table 8.4: Client Side CPU Utilization

Table 8.4 shows that the offloading is complete in the sense that there are no components left on the host processor. An idle machine and a machine that is running the fully offloaded client both consume the same background level of CPU cycles. The non-offloaded user-space client consumes more CPU, although the load is very small compared to the total capability of the host processor.

In terms of L2 cache misses, the idle machine and offloaded client have the same count, while the non-offloaded client generates 12% more misses. Much of this is due to the MPEG decoding process.

8.2 Total Ordering

Total Order (TO) algorithms have been extensively studied in the literature [21]. A TO algorithm is a fundamental building block in the construction of distributed fault-tolerant applications. They are typically used to provide a communication primitive that allows processes to agree on the set of messages they deliver and also on their delivery order. Total ordering is particularly useful for implementing fault-tolerant services, database replication and locking services [4]. A TO algorithm that assumes an unreliable failure detector is equivalent to the consensus problem [13]. It has been shown that consensus cannot be solved in this type of systems in fewer than two communication steps [37]. Many TO algorithms for asynchronous systems use consensus as a building block, but the implementation can be expensive both in terms of communication steps and number of messages exchanged between hosts. This overhead is further exacerbated if in addition to the TO algorithm, the host also executes a resource-demanding application such as a typical High Performance Computing (HPC) application.

Offloading a TO algorithm, either in full or for particular components, can greatly improve the performance of distributed applications for several reasons. First, a TO algorithm packaged as an Offcode can be easily reused by a variety of applications. Second, the reduced load on the host machine will improve the performance of such applications; and third, an offloaded TO may take advantage of specific hardware capabilities in order to improve its overall performance. For example, as shown in the traffic generator example (Section 8.3), the small dispersion of the inter-arrival times of ethernet packets may be used to implement better accurate failure detectors and to maintain finer-grained timeouts for message retransmissions. Another possibility is to use hardware-based encryption engines, which are found on several computer peripheral devices, in order to support Byzantine models.

8.2.1 Offload-Aware TO Architecture

A simple offload-aware total order application has been designed and implemented for our NIC (the application uses the NICOS framework discussed in Chapter 6). To simplify the proof of concept implementation, we assume that there are no physical link disconnections, switch failures, or process or node crashes. We do not assume a reliable message transmission—messages can be lost due to buffer overflow at the NIC, host or switch. The sample application is comprised of several components that appear on the left side of Figure 8.6.

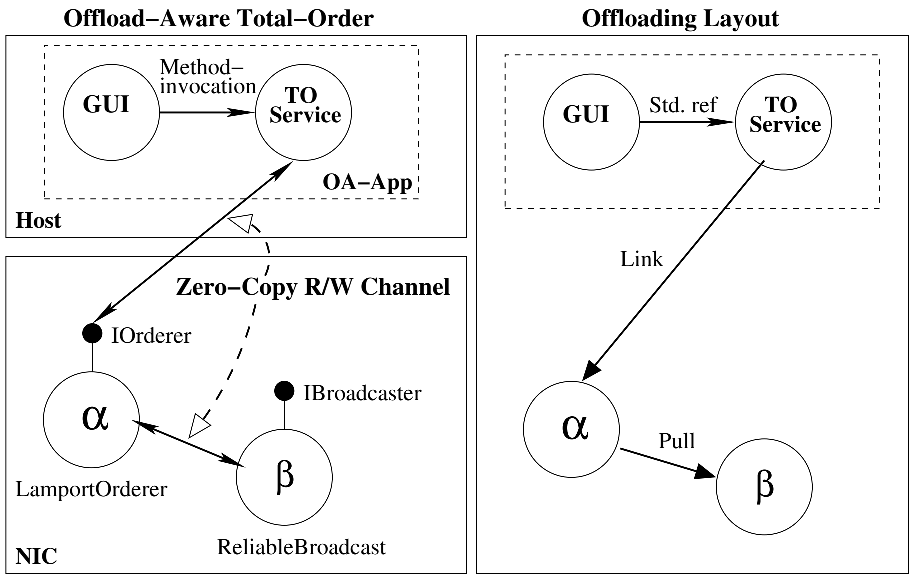


Figure 8.6: Total-Order Offload Architecture

1. *GUI*: The Graphical User Interface controls the TO application. It enables the user to define the rate at which messages are transmitted and their size. The GUI presents the message order once it is determined.

2. *TO Service*: The TO Service is an application library used by the GUI, that in turn uses the Offcode to provide two basic total-order APIs: *TO_Broadcast* and *TO_Receive*. The first API broadcasts a message and the second receives the next message for which the TO has been established.

3. *LamportOrderer*: This Offcode (denoted by the letter α) presents the *IOrderer* interface that implements a TO algorithm. Specifically, we have implemented the Lamport's Timestamp

ordering algorithm [40]. This Offcode interacts with the *TO Service* in a well defined interface, discussed below.

4. *ReliableBroadcast*: This Offcode (denoted by β), provides the reliable broadcast service that is needed by the *LamportOrderer* Offcode. In our implementation, multicast is used in order to efficiently send messages to peer hosts. Albeit our simplifying assumptions, message omissions may still occur due to buffer overflow. To address this issue, this component implements a simple negative acknowledgment scheme. Once a missing message is detected (indicated by a "hole" in the message sequence numbers), the receiving node periodically sends a retransmit request to the sending source of the missing message. Once the message's source receives the retransmit request, it multicasts the message.

The left side of Figure 8.6 also indicates the HYDRA communication channels that are used. A reliable unicast channel with a zero-copy policy for read and write is used in order to eliminate the OS networking stack overhead. Basically, the *TO Service* manages the application's memory descriptors and effectively determines the control-flow policies of the application (descriptors for received messages are also posted by this component). In order to send a message, the *TO Service* creates a *Call* object and invokes the channel. The NIC-resident HYDRA runtime DMAs the message and notifies the *LamportOrderer* Offcode that a new message should be transmitted. The "orderer" Offcode timestamps the message and multicasts it using the *Broadcaster* interface, which is implemented by the *ReliableBroadcast* Offcode.

Received packets are first handled by the *ReliableBroadcast* Offcode. The Offcode is operated in two phases: At the first phase, the Offcode transfers the received packet to a pre-posted descriptor at the host using DMA. Note that the message cannot be delivered to the application yet, since the message order has not been determined. Because the NIC has a small amount of memory, it is better to release the NIC's memory as soon as possible. The message identifier and timestamp are the only data that is saved on the NIC by the *LamportOrderer* Offcode. The second phase begins once the message order has been determined by the TO algorithm. The *LamportOrderer* Offcode creates a *Call* with the messages' order and invokes the channel connected to the *TO Service*. Once

Nodes	Hardware-based TO		Offload-Aware TO	
	Throughput [Mbps]	Latency [ms]	Throughput [Mbps]	Latency [ms]
3	310.5	4.2	301.8	8.7
5	362.5	4.1	324.6	9.5

Table 8.5: TO Performance (all-to-all)

the order is known at the *TO Service* component, the ordered messages can safely be delivered to the application.

The right side of Figure 8.6 presents the offloading layout that is designed by the developer. The *GUI* holds a standard reference to the *TO Service* component. This component holds a *Link* reference to the "orderer" components α, since it has no special offloading constraints. On the other hand, the "orderer" Offcode must be offloaded *with* the broadcast Offcode (i.e, β) hence a *Pull* constraint is used. Note that in order to compare the results of this offload-aware TO algorithm with a non-offloaded version, a developer merely needs to interchange the two constraints and re-execute the application. The effect of doing so is that the "orderer" will be executed at the host while the broadcaster remains at the NIC.

8.2.2 Total Ordering Evaluation

The benchmark consists of five Intel Pentium 4 2.4 GHz systems, with 512MB of RAM and 32-bit, 33 MHz PCI bus. Each machine was equipped with programmable Netgear 620 NICs, which have 512 kB of memory. Each host executed Linux OS version 2.6.11 with the HYDRA module enabled. The hosts were interconnected by a Gigabit ethernet switch (Dell PowerConnect 6024). The right side of Table 8.5 presents the maximum throughput and latency measurements for the offload-aware TO when all nodes act as both senders and receivers. Each node generate traffic at a rate bounded by the flow control mechanism imposed by the *TO Service* component. The presented latency is defined as the time elapsed between the *TO_Broadcast* and *TO_Receive* method invocations that refer to the same message.

The benchmark results have been compared with those from a recent work by Dolev et al. [5], which are given on the left side of the table with title "Hardware-based TO". That work implements a wire-speed total order algorithm using hardware-based component comprised of two switches connected back-to-back. Each host is equipped with two NICs: one NIC is used for transmitting (connected to the first switch) and one for receiving (connected to the second switch). The back-to-back switch connection serializes the packets, thus effectively acts as a hardware sequencer. In addition to the switch configuration, a lightweight user-space TO algorithm is invoked at each node.

The *throughput* obtained from the offload-aware TO application is close to that of the hardware-based solution. Note that the throughput increases with the number of nodes due to PCI bus properties as explained in previous work [5, 71].

Although with HYDRA we have used a *software* algorithm to order the messages we found that bypassing the OS networking stack overhead enabled us to significantly increase the throughput over typical user-based total ordering. This fact strengthens the motivation for offloading and specifically for using HYDRA.

As for the measured latency, the results are approximately twice those in the hardware-based configuration. Although the ordering algorithm is offloaded to the NIC, a distributed solution requires an extra round of communication that is not required in centralized solutions (like the hardware-based solution). In addition, Lamport's timestamp algorithm is known to be very expensive in terms of communication overhead and latency; messages must be received from every node in order to be able to determine the messages' order. Other ordering algorithms can reduce this overhead.

8.3 Traffic Generator

Generating steady network traffic at high rates is difficult given the variety of sources of delays and unpredictability in a modern computer system, including interrupts from devices, cache and TLB misses, and power management changes. This section presents an offload-aware traffic generator that produces a packet stream with fixed inter-packet delays. The offloaded traffic generator is evaluated and compared with an equivalent user-level application.

The traffic generator is comprised of two components: a GUI that is used by the user to configure the traffic attributes, and a *StreamGenerator* component that generates the stream of packets given user settings on protocol type, length, ports, inter-packet delay, burst size, etc. The *Stream-Generator* component is designed as an offcode. The GUI is the offcode's controller and creates a specialized, zero-copy, channel for communication. The APIs for interaction between the GUI and the *StreamGenerator* offcode ore omitted here for brevity, as are details of the offcode description file.

The traffic generator is implemented twice: once using HYDRA and once without the use of an offloaded component. The benchmark consists of two hosts, Intel Pentium 4 2.4 GHz with 512 MB and a Tigon2 programmable network card, interconnected by a 100 Mb/s switch. The link capacity is fully utilized by generating packets at fixed inter-packet delays and for different frame sizes.

8.3.1 User-Space Traffic Generator

The benchmark results for the user-space application are given in Table 8.6. Although the achieved throughput is quite good, the dispersion of the inter-arrival times is enormous, so large as to make the average almost meaningless. Figure 8.7 shows the cumulative distribution function (CDF) for three packet sizes to better display the distribution of arrival times and illustrate the wide dispersion in these measurements.

Size	*Tput*	*Avg. Arrival* $\pm$ *Std*	*CPU* $\pm$ *Std*
Bytes	*Mb/s*	μs	*%*
64	6.0	140 $\pm$ 8 000	100 $\pm$ 3
80	13.4	141 $\pm$ 9 000	99 $\pm$ 7
96	21.8	159 $\pm$ 11 000	99 $\pm$ 8
192	56.8	164 $\pm$ 6 000	98 $\pm$ 11
384	96.7	175 $\pm$ 4 000	81 $\pm$ 11
768	97.8	205 $\pm$ 4 000	37 $\pm$ 28
1514	98.6	244 $\pm$ 5 000	33 $\pm$ 5

Table 8.6: User Space Traffic Results

It is also evident from the table that delivering the generated data to the application is difficult due to the very high CPU load, especially with small packet sizes. The processor capacity problem, driven by the costs associated with interrupts, directly impacts the throughput seen by the applications. As an example, the calculated inter-arrival times for 1500 byte ethernet frames is approximately 120 μs for 100 Mb/s, 12 μs for 1 Gb/s and 1.2 μs for 10 Gb/s ethernet. The

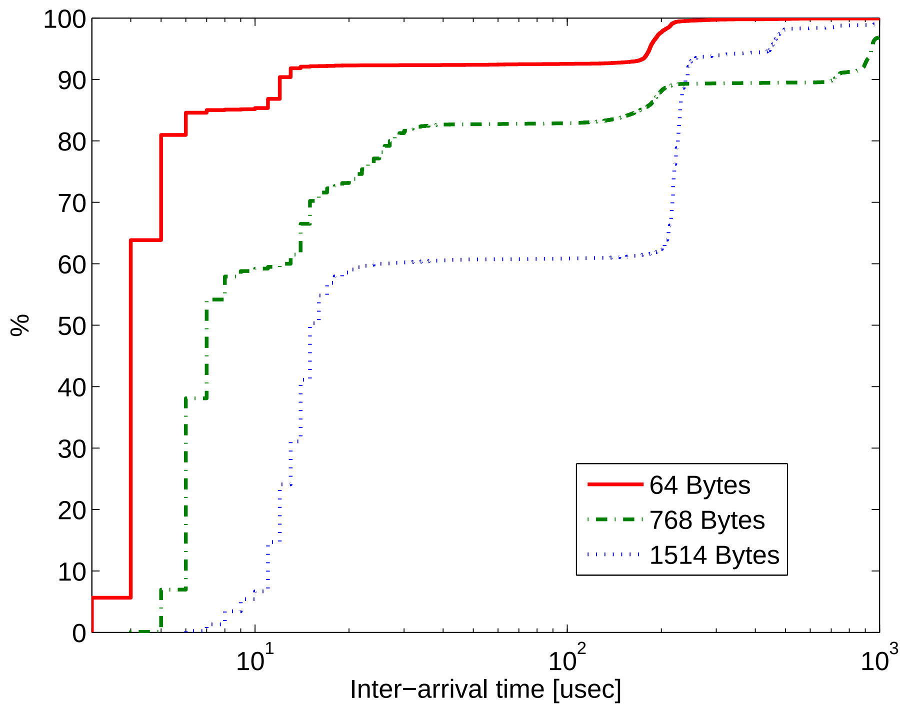


Figure 8.7: User-Space Traffic Distribution

observed interrupt overhead for an empty interrupt handler is between 5–10μs, consuming all but only 17% of the total available CPU cycles.

Size	*Tput*	*Avg. Arrival* $\pm$ *Std*	*CPU*
Bytes	*Mb/s*	μs	*%*
64	23.9	34 $\pm$ 6	2
64*	51.5	16 $\pm$ 8	2
768	98.4	65 $\pm$ 13	2
1514	98.8	126 $\pm$ 50	2

Table 8.7: Offload-Aware Traffic Results

8.3.2 Offload-Aware Traffic Generator

The results from the offload-aware traffic generator are summarized in Table 8.7 and shown as a CDF in Figure 8.8. For both tests, in order to accurately measure the throughput and the inter-arrival times, a second NIC with a simple traffic analyzer offcode has been used. The data shows that the inter-arrival times are uniform with small standard deviation. The sharp vertical edges in the CDF indicate that the majority of the packets arrived within the same expected inter-arrival

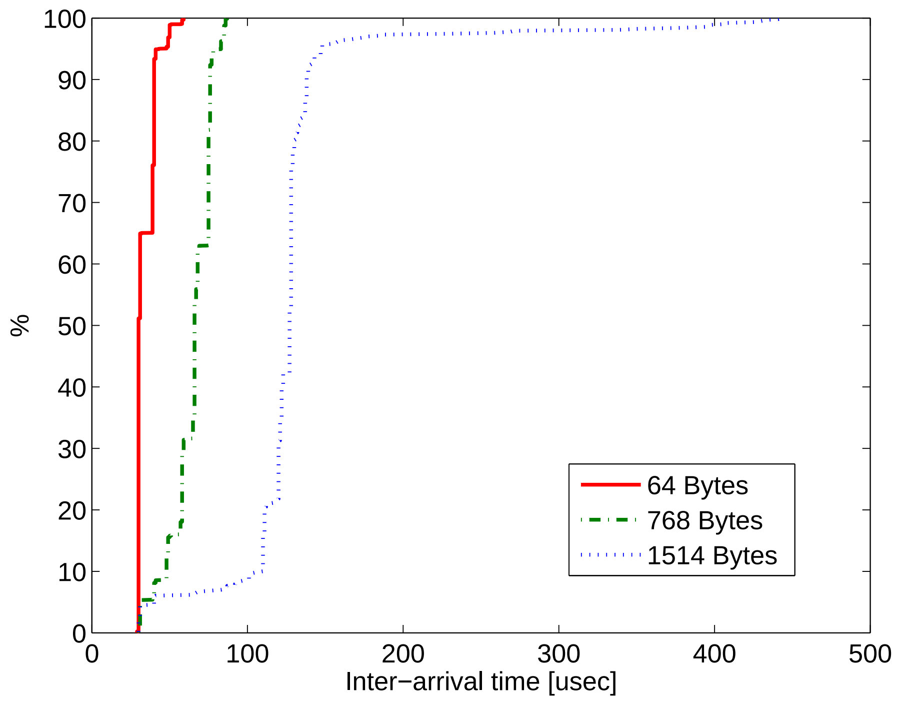


Figure 8.8: Offload-Aware Traffic Distribution

time. Notice that for 64-byte packets, the achieved throughput is only a quarter of the link's bandwidth. In order to achieve the full link capacity, a generator must produce a 64-byte packet approximately every 5 μs. Since the runtime is not optimized for this or any specific application, the generator can only send packets at a rate limited by the device's OS constraints, which in this case is limited by the number of MAC descriptors at the NIC and the processing overhead involved in managing them. In order to further improve the throughput for such small packets, an optimized version of the device's OS has been implemented. This modified version can reuse a single MAC descriptor for sending the same packet multiple times. The table shows that for the optimized version (indicated by the $64^{\star}$ table entry) the throughput has been significantly improved. This sort of optimization may be undertaken as needed by particular applications that use HYDRA.

8.4 Offloaded Firewall

An application of particular promise for offloading is a network firewall. Performing packet filtering closer to the network ingress can significantly improve overall capability by freeing the host processor to perform other network activities such as forwarding. A firewall application on a NIC also has the additional advantage that it adds an interesting level of complexity to intruders who would attempt to attack the filtering system.

We have designed and implemented a firewall application (henceforth called: SCIRON) that is implemented as a set of offcodes that perform basic rules. Rules can be dynamically created or removed by the firewall controller which is executed at the host.

8.4.1 Overview and Motivation

Offloading firewall logic to a NIC offers several benefits. First, an offloaded firewall is an OS independent implementation. Second, it is harder to tamper with hardware as opposed to a software implementation. Firewall applications are computationally expensive for several reasons:

- The host's CPU is repeatedly interrupted by the NIC on incoming packets. The processing power required to handle the interrupts is wasted if the packet is doomed to be discarded.
- An adversary can try to perform a denial of service (DoS) attack by sending packets from many computers, in an attempt to overload the system.
- The networking stack has significant overhead.
- The PCI [2] bus is a major bottleneck especially in today's incline towards faster networking fabrics.

8.4.2 SCIRON Architecture

This section presents the main components of SCIRON runtime. The runtime is comprised of two main components: The SCIRON enforcement module and SCIRON's management console.

The enforcement module is the engine of SCIRON's firewall that actively enforces the security policy upon incoming and outgoing packets. SCIRON's firewall is an ordered 5-tuple firewall. When a packet arrives, a sequential pass over the rules is performed. The action (accept or reject) associated with the first rule that matches the packet header is performed. If there is no match, the default policy action (reject all) is performed.

Scenario	Kernel based Firewall		NIC based Firewall	
	CPU Load	Throughput [Mbps]	CPU Load	Throughput [Mbps]
100% Discard	78.96%	0	0.04%	0
50% Discard	93.02%	19	10.43%	43

Table 8.8: Firewall Performance

SCIRON's management console provides remote administration and logging capabilities. Administrators can remotely install security policies at enforcement modules of machines in their domain. This is done by communicating with SCIRON's embedded enforcement module using a proprietary protocol called *SRPP* (SCIRON Remote Policy Protocol).

An administrator can also determine the policy for monitoring and logging events to the management console. This is done by marking specific rules as *log-rules*. Packets caught by these rules will generate a log packet containing the packet's information. The logged packet is then sent to the management console. Allowing real-time monitoring and tracking of the network activities, enables the administrator to immediately act upon potential attacks.

SCIRON management console is comprised of the following modules: (1) Management console GUI - a tool used for defining and managing the security policy; (2) Log viewer - A server application which receives log packets sent by the various enforcement modules and displays them graphically to the administrator; (3) Policy builder - a tool for verifying the correctness of the security policy defined by the administrator, by searching for shadowed and redundancy rules. The verifier implements the algorithm presented in [3]

8.4.3 SCIRON Evaluation

In order to simulate common kernel-based firewalls for performance evaluation, we have also implemented the firewall at the kernel. All comparisons shown below compare the same firewall code, with the same filtering policy, between the kernel-based firewall and the offload-aware firewall.

Performance can be measured using two parameters. The first is the load on the CPU and the second is the throughput. In this section we discuss several typical scenarios. In the first scenario, shown in the top row of Table 8.8, the firewall simply discards all packets. During this scenario the CPU is only running system processes. As we expect, in this scenario the CPU utilization when using the firewall implemented on the NIC is approximately zero (0.04%), whilst for the same firewall on the host it is quite high (78.96%).

The second scenario presented is shown in the bottom row of Table 8.8 where half of the traffic is discarded randomly. It is evident again that the NIC based firewall has much better performance both in CPU utilization and throughput. The benchmark machine was the same in both cases, an Intel Pentium 4 CPU at 2.4 GHz with 512 MB of memory and 100 Mb/s ethernet.

The results clearly show that offloading firewall logic to a NIC has many advantages. In scenarios with a heavy incoming packet load (especially if packets need to be discarded) a firewall offloaded to a NIC significantly improves both CPU utilization as well as packet throughput. On the less likely scenarios of heavy outgoing packet traffic, offloading firewall logic to a NIC is slower than conventional firewalls. It is important to note that our implementation is based on an obsolete NIC. We expect that the performance gain will be more pronounced when utilizing an advanced NIC. Although current NICs hardware is continuously improving, the host CPU speed will likely continue to be faster than NIC hardware. In order to further improve the sending flow performance, a mixed paradigm can be used. In this model, the processing of outgoing packet is performed at the host while the incoming packets are processed in the NIC.

Chapter 9

Conclusions and Future work

Hardware and software are neck and neck, pushing each other forward. This research claims that it is the OS's turn to act. Hardware manufacturers have provided an excessive amount of computing resources, which are just sitting there idling most of the time. It is time for the OS community to design tools and programming abstracts that will enable a developer to efficiently utilize every programmable component in the system.

In this chapter we summarize the contribution of this work, outline the conclusions and describe ongoing and future work.

9.1 Contribution

This research presented the HYDRA framework [75, 73, 76, 77] which proposes a unique new dimension of flexibility for the architects of high performance applications: the ability to program offloading layout policies separately from the application's logic. HYDRA defines a programming model that carefully balances between programmer scalability and system scalability. As programmable devices will continue to grow in popularity, it is only a matter of time until an OS on a workstation or a PC will be considered as a switching element among heterogenous processing cores.

The contributions of the research described in this dissertation are twofold: 1. A programming model that enables the developer to encode applications capable of utilizing programmable devices by specifying an offloading layout; and 2. an effective runtime infrastructure that realize the model and supports it efficiently.

9.2 Ongoing Work

This research is an ongoing effort. The natural evolution of this framework is in supporting kernel level components, such as device drivers. As the problem of device drivers reliability is acute, it should be our first target. Harnessing the existing power of peripheral devices to offload and isolate device drivers may drastically improve the overall system's reliability and dependability. Pushing drivers down into the peripheral itself could also simplify current kernels. Each peripheral would be required to emulate a standard "virtualized" device interface, thus the kernel would only support one device interface per peripheral type. One for storage devices, one for network devices, one for graphics cards, etc. In "this world", kernel developers would focus on optimizing **application** support, **not** on **device** support. Device vendors would then be free to optimize and improve their embedded implementations. New operating system versions would be easier to deploy because they could be tested on a standard device instead of all reasonably current devices. In addition to that, from a security perspective, implementing device drivers in the peripheral will make them harder to attack and penetrate.

Bibliography

[1] Killer NIC. Homepage at http://www.killernic.com/KillerNic/.

[2] PCISIG industry organization, PCI specification. http://www.pcisig.com/specifications.

[3] Ehab S. Al-Shaer and Hazem H. Hamed. Discovery of policy anomalies in distributed firewalls. In *INFOCOM*, 2004.

[4] Yair Amir and Ciprian Tutu. From total order to database replication. In *ICDCS '02*. IEEE Computer Society.

[5] Tal Anker, Danny Dolev, Gregory Greenman, and Ilya Shnayderman. Wire-speed total order. In *IPDPS'06*.

[6] Muli Ben-Yehuda, Jon Mason, Jimi Xenidis, Orran Krieger, Leendert van Doorn, Jun Nakajima, Asit Mallick, and Elsie Wahlig. Utilizing iommus for virtualization in linux and xen. In *OLS '06: The 2006 Ottawa Linux Symposium*, pages 71–86, July 2006.

[7] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *15th SOSP*, 1995.

[8] Nathan L. Binkert, Ali G. Saidi, and Steven K. Reinhardt. Integrated network interfaces for high-bandwidth TCP/IP. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 315–324, New York, NY, USA, 2006. ACM Press.

[9] A. Birell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[10] Ken Birman, Robert Constable, Mark Hayden, Jason Hickey, Christoph Kreitz, Robbert van Renesse, Ohad Rodeh, and Werner Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 149–161, Hilton Head, SC, 2000. IEEE Computer Society Press.

[11] N. Brown and C. Kindel. *Distributed Component Object Model Protocol — DCOM/1.0. Internet Draft*, January 1998. Available at http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-02.txt.

[12] Matthew Burnside and Angelos D. Keromytis. High-speed i/o: the operating system as a signalling mechanism. In *NICELI '03: Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence*, pages 220–227, New York, NY, USA, 2003. ACM Press.

[13] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, pages 225–267, 1996.

[14] S.-C. Cheng, J.-A. Stankovic, and K. Ramamritham. Scheduling algorithms for hard real-time systems: a brief survey. pages 150–173, 1989.

[15] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 73–88, New York, NY, USA, 2001. ACM Press.

[16] ClearSpeed. Clearspeed advance. Available at site: http://www.clearspeed.com/products/cs_advance/.

[17] "Microsoft Corporation". "scalable networking: Network protocol offload introducing tcp chimney". *WinHEC*, 2004.

[18] Andy Currid. TCP offload to the rescue. *Queue*, 2(3):58–65, 2004.

[19] Ariel Daliot and Danny Dolev. Self-stabilization of Byzantine protocols. In *Proceedings of 7th Symposium on Self-Stabilizing Systems (SSS'05)*, Barcelona, 2005.

[20] Ariel Daliot and Danny Dolev. Self-stabilizing Byzantine agreement. In *Twenty-fifth ACM Symposium on Principles of Distributed Computing (PODC'06)*, Denver, Colorad, July 2006.

[21] X. Defago, A. Schiper, and P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[22] Suru Dissanaike, Pierre Wijkman, and Mitra Wijkman. Utilizing xml-rpc or soap on an embedded system. In *ICDCS Workshops*, pages 438–440, 2004.

[23] Shlomi Dolev and Jennifer L. Welch. Wait-free clock synchronization. *Algorithmica*, 18(4):486–511, 1997.

[24] Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of Byzantine faults. *Journal of the ACM*, 51(5):780–799, 2004.

[25] Marc E. Fiuczynski, Richard P. Martin, Tsutomu Owa, and Brian N. Bershad. Spine: a safe programmable and integrated network environment. In *EW 8*, 1998.

[26] A. P. Foong, T. R. Huff, H. H. Hum, J. R. Patwardhan, and G. J. Regnier. TCP performance re-visited. In *ISPASS '03: Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 70–79, Washington, DC, USA, 2003. IEEE Computer Society.

[27] Alessandro Forin, Johannes Helander, Paul Pham, and Jagadeeswaran Rajendiran. Component based invisible computing.

[28] Maxim Garbarnik. Dynamic Offloading Infrastructure for Programmable Devices. Master's thesis, The Hebrew University Of Jerusalem, Israel., October 2007.

[29] P. Gilfeather and A. B. Maccabe. Modeling protocol offload for message-oriented communication. In *Cluster 2005*, 2005.

[30] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: High performance graphics coprocessor sorting for large database management. In *ICMD*, June 2006.

[31] J. Helander and A. Forin. Mmlite: A highly componentized system architecture, 1998.

[32] O. Holder, I. Ben-Shaul, and H. Gazit. System support for dynamic layout of distributed applications. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99)*, pages 163–173, Austin, TX, May 1999.

[33] Ophir Holder, Israel Ben-Shaul, and Hovav Gazit. Dynamic layout of distributed applications in FarGo. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 163–173, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[34] J. Holliday. Replicated database recovery using multicast communication. In *Proceedings of the Symposium on Network Computing and Applications (NCA'01)*, Cambridge, MA, 2001. IEEE.

[35] Larry Huston, Rahul Sukthankar, Rajiv Wickremesinghe, M. Satyanarayanan, et al. Diamond: A storage architecture for early discard in interactive search. In *FAST'04*.

[36] IBM. IBM System z cryptography for highly secure transactions. Available at site: `http://www-03.ibm.com/systems/z/security/cryptography.html`.

[37] Keidar and Rajsbaum. On the cost of fault-tolerant consensus when there are no faults. *SIGACT News*, 2001.

[38] Bettina Kemme and Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Computer Systems*, 25(3):333–379, September 2000.

[39] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[40] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[41] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time envi ronment. *Journal of the ACM*, 20(1):46–61, 1973.

[42] Panagiotis Louridas. Soap and web services. *IEEE Software*, 23(6):62–67, 2006.

[43] Arthur B. Maccabe, Wenbin Zhu, Jim Otto, and Rolf Riesen. Experience in offloading protocol processing to a programmable NIC. In *IEEE ICCC*, 2002.

[44] Inc. Sun Microsystems. RPC: Remote procedure call. Proposal RFC1050, Internet Engineering Task Force, April 1988.

[45] Jeffrey C. Mogul. TCP offload is a dumb idea whose time has come. In *HotOS*, pages 25–30, 2003.

[46] Stephen James Muir. *Piglet: an operating system for network appliances*. PhD thesis, 2001. Supervisor-Jonathan M. Smith.

[47] Adamovsky Ola. ILP Formulation for Dynamic Offloading Layouts. Master's thesis, The Hebrew University Of Jerusalem, Israel, October 2007.

[48] OpenGroup. DRC announcement at theregister. Available at site: `http://www.drccomputer.com`.

[49] Physx. Ageia physx. Available at site: `http://www.ageia.com/physx/index.html`.

[50] Ian Pratt and Keir Fraser. Arsenic: A user-accessible gigabit ethernet interface. In *INFOCOM*, pages 67–76, 2001.

[51] RDMA Consortium. Architectural specifications for RDMA over TCP/IP. `http://www.rdmaconsortium.org/`.

[52] R. Recio, P. Culley, D. Garcia, J. Hilland, and B. Metzler. An RDMA protocol specification. http://www.ietf.org/internet-drafts/draft-ietf-rddp-rdmap-04.txt, 2005.

[53] Greg Regnier, Srihari Makineni, Ramesh Illikkal, Ravi Iyer, Dave Minturn, Ram Huggahalli, Don Newell, Linda Cline, and Annie Foong. TCP onloading for data center servers. *Computer*, 37(11):48–58, 2004.

[54] Greg Regnier, Dave Minturn, Gary McAlpine, Vikram A. Saletore, and Annie Foong. Eta: Experience with an intel xeon processor as a packet processing engine. *IEEE Micro*, 24(1):24–31, 2004.

[55] E. Riedel and G. Gibson. Active disks - remote execution for network-attached storage, 1997.

[56] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. In *VLDB'90*.

[57] R. Riggs, J. Waldo, and A. Wollrath. Pickling state in the Java system. In *Proceedings of the USENIX Second International Conference on Object-Oriented Technologies (COOTS'96)*, Ontario, Canada, June 1996.

[58] Piyush Shivam, Pete Wyckoff, and Dhabaleswar Panda. EMP: zero-copy OS-bypass NIC-driven gigabit ethernet message passing. In *SC'01*, November 2001.

[59] Edi Shmueli and Dror G. Feitelson. Backfilling with lookahead to optimize the performance of parallel job scheduling. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 228–251. Springer Verlag, 2003. Lect. Notes Comput. Sci. vol. 2862.

[60] Jon Siegel. OMG overview: CORBA and the OMA in enterprise computing. *Commun. ACM*, 41(10):37–43, 1998.

[61] Sun Microsystems, Inc. *Java Remote Method Invocation (RMI) Specification*, October 1998. Available at `http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html`.

[62] Sun Microsystems Inc. *Java Remote Method Invocation Security Extension (draft)*, 1999. Available at: `http://java.sun.com/products/jdk/rmi/`.

[63] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. Nooks: an architecture for reliable device drivers. In *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop: beyond the PC*, pages 102–107, New York, NY, USA, 2002. ACM Press.

[64] TiVo Inc homepage. Available at site: `http://www.tivo.com`.

[65] Dan Tsafrir, Yoav Etsion, Dror G. Feitelson, and Scott Kirkpatrick. System noise, OS clock ticks, and fine-grained parallel applications. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 303–312, New York, NY, USA, 2005. ACM Press.

[66] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

[67] R. van Engelen and K. Gallivan. The gsoap toolkit for web services and peer-to-peer computing networks, 2002.

[68] P. Veríssimo and A. Casimiro. The Timely Computing Base model and architecture. *IEEE Transactions on Computers*, 51(8), 2002.

[69] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.

[70] w3 organization. Web Service Definition Language (WSDL). http://www.w3.org/TR/wsdl.

[71] W. Wadge. Achieving gigabit performance on programmable ethernet network interface cards. B.Sc. Final Report, University of Malta, 2001.

[72] Ralph O. Weber. Information technology—SCSI object-based storage device commands -2 (OSD-2). Technical Report T10/1731-D, INCITS Technical Committee T10, October 2004.

[73] Yaron Weinsberg, Tal Anker, Danny Dolev, and Scott Kirkpatrick. On a NIC's operating system, schedulers and high-performance networking applications. In *HPCC-06*.

[74] Yaron Weinsberg and Israel Ben-Shaul. A programming model and system support for disconnected-aware applications on resource-constrained devices. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 374–384, New York, NY, USA, 2002. ACM Press.

[75] Yaron Weinsberg, Danny Dolev, Pete Wyckoff, and Tal Anker. Accelerating distributed computing applications using a network offloading framework. In *IPDPS'07*.

[76] Yaron Weinsberg, Danny Dolev, Pete Wyckoff, and Tal Anker. Hydra: A novel framework for making high-performance computing offload capable. In *LCN'06*.

[77] Yaron Weinsberg, Elan Pavlov, Yossi Amir, Gilad Gat, and Sharon Wulff. Putting it on the NIC: A case study on application offloading to a Network Interface Card (NIC). In *IEEE CCNC'06*.

[78] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), 1995.

[79] World Wide Web Consortium (W3C). Simple Object Access Protocol (SOAP) 1.1. Available at site: `http://www.w3.org/TR/soap`.

[80] World Wide Web Organization. Web Services Activity. Available at: `http://www.w3.org/2002/ws/`.

[81] XML-RPC Specification Homepage. XML-RPC Specification. Available at site: `http://www.xmlrpc.com/spec`.

[82] Yaron Weinsberg, Shimrit Tzur-David, Tal Anker and Danny Dolev. High performance string matching algorithm for a Network Intrusion Prevention System (NIPS), 2006.